目　　录

第1章　DSP开发系统

本章介绍了几种数字信号处理（DSP）开发工具，包括最常用的程序代码编辑调试软件（CCS）和DSP初学者工具包（DSK）。CCS提供了集成开发环境（IDE），而DSK电路板上有TMS320C6711浮点处理器，能完全支持电路系统输入输出信号。另外，随DSK还附带了三个测试软件和硬件程序实例。通过本章的学习，要学会使用CCS、TM320C6711 DSK，并学会使用它们和程序实例来测试软硬件工具。

1.1　引言

数字信号处理器，如TMS320C6x（C6x）系列处理器，和高速专用微处理器一样，具有适合于信号处理应用的特殊结构和指令集。C6x用来表示德州仪器公司（TI公司）TMS320C6000系列的数字信号处理器，C6x数字信号处理器的结构非常适合于高强度的数学运算。C6x基于超长指令字（VLIW）结构，被认为是TI公司运算能力最强的处理器之一。

数字信号处理器具有广泛的应用，从通信、控制到语音、图像处理等场合都可以看到DSP的影子，在蜂窝移动电话、传真机、调制解调器、磁盘驱动器、收音机等设备中，都可以发现有DSP的应用。由于这些处理器具有较高的性价比，因此已被广泛应用于消费类产品中。另一方面，处理器能很方便地重新编程以适应不同的应用要求，因此它们能够处理不同的任务。此外由于软件和硬件开发支持成本较低，因此DSP技术取得了较大的成功。例如利用DSP技术，调制解调器和语音识别的成本就可能非常便宜。

DSP处理器主要应用于实时信号处理。实时信号处理意味着处理过程必须与外部事件保持同步，而非实时信号处理就不会有这样的限制。需要实时处理的外部事件一般是模拟信号。尽管由电阻等分立电子元件组成的模拟系统对温度变化特别敏感，但DSP系统受外界环境（如温度等）的影响却很小。DSP处理器除具有微处理器的优点外，还具有易于使用、方便灵活和价格便宜等优点。

现在，很多书籍和文章都介绍了DSP在多种应用中的重要性[1-20]。从甚高频通信的光纤传输到最适合音频信号处理的DSP，很多技术已经用来进行实时处理。处理器常用于频率范围从0到20 kHz信号的实时处理。语音信号可用8 kHz（采样速度）抽样频率进行抽样，这意味着每一个抽样点的抽样时间是1/8 kHz或0.125 ms，而CD的抽样频率通常是44.1 kHz。现在，抽样频率高达兆赫兹量级的A/D采集板也能够买到。

最基本的应用系统通常有一个模数转换器（ADC），用来捕获模拟信号。当模拟信号转换成数字信号后，信号再经过C6x的DSP处理，最后进入数模转换器（DAC），变成模拟信号进行输出。最基本的应用系统通常还含有一个输入抗混叠滤波器和一个输出滤波器，它们分别用来滤除不需要的带外信号或平滑、重构处理过的输出模拟信号。

1.2　DSK支持工具

本书的大部分工作涉及DSP应用程序设计和实现。为了进行实验，需要用到下面的工具：

1. TI公司的DSP初学者工具包（DSK）。DSK包括：
   (a) 程序代码编辑调试软件（CCS），它提供了必要的软件支持工具和集C编译器、汇编器、连接器、调试器等于一体的集成开发环境（IDE）。
   (b) DSK电路板，如图1.1所示，DSK电路板上有TMS320C6711（C6711）浮点数字信号处理器和支持16位输入输出的编解码器。
   (c) 连接DSK电路板和PC的并行接口电缆（DB25）。
   (d) DSK电路板电源。

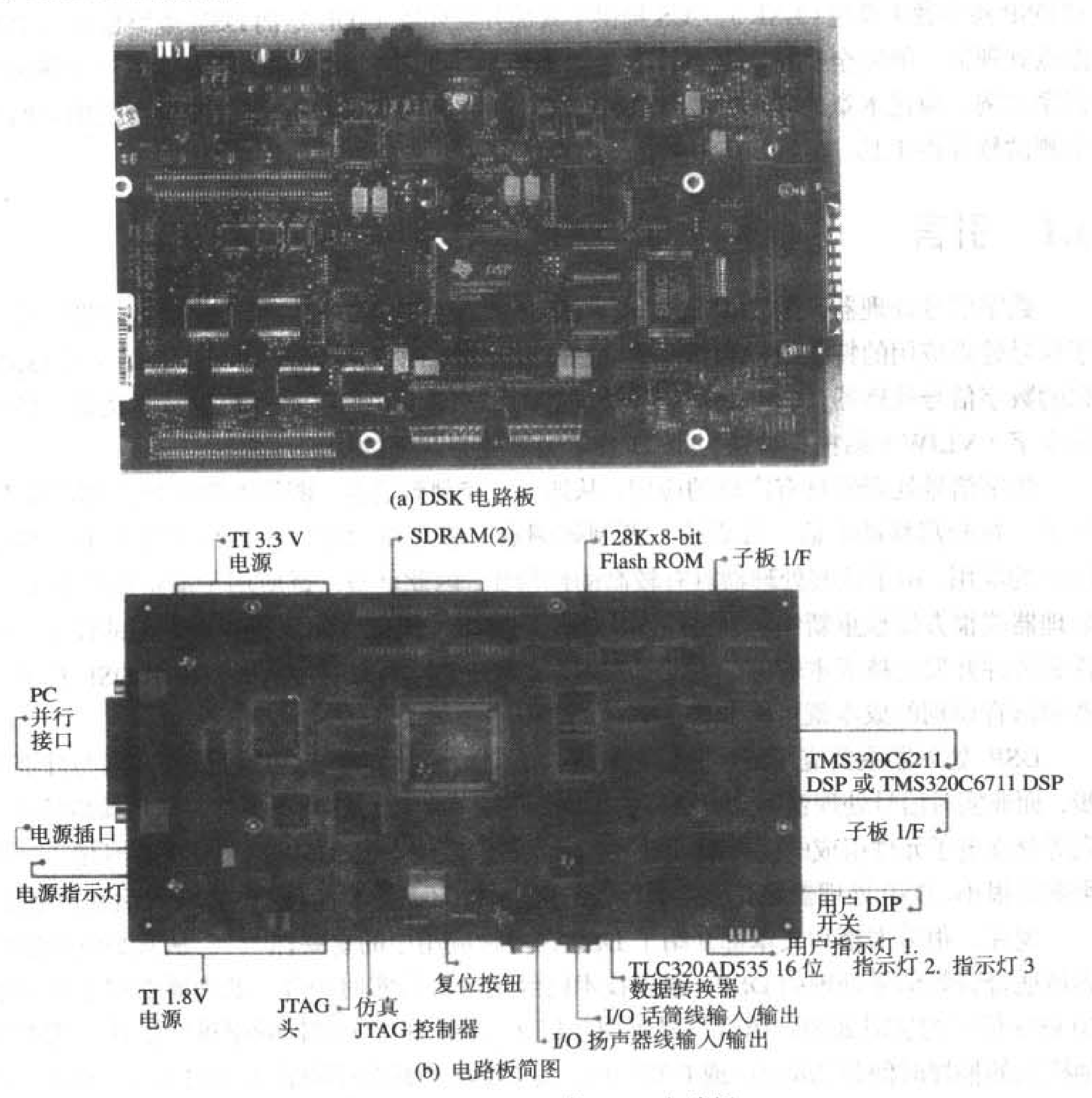


图1.1 基于TMS320C6711的DSK电路板

2. 一台IBM兼容PC，DSK电路板通过DSK工具包中的DB25电缆与PC的并行接口相连。
3. 示波器、信号发生器、扬声器各一台，也可选用信号分析和频谱分析仪器。可使用共享软件，利用PC和声卡构建如示波器、函数发生器或频谱分析仪等虚拟仪器。

除第9章的学生课程设计文件外，本书列出和讨论的所有文件和程序都可以从www.hxedu.com.cn或www.phei.com.cn下载[①]。大部分程序实例还能够在定点C6211 DSK（现在已经停产了）上运行。

① 在本书中，所有从这两个网站下载的文件和程序统都称为辅助材料——编者注。

1.2.1 DSK 电路板

DSK 工具包的功能很强而价格相对较便宜（目前为 295 美元），它是一个完整的 DSP 系统，包括了实时信号处理必要的硬件和软件支持工具[21-33]。DSK 电路板的尺寸大约为 5×8 英寸①，上面有 C6711 浮点数字信号处理器[22]和 16 位的 AD535 输入输出编解码器。

DSK 板上的 AD535 编解码器采用 Sigma-Delta 技术进行模数和数模转换，利用 DSK 板上 4 MHz 的时钟产生固定的 8 kHz 的抽样时钟。

DSK 板上还有一个子板扩展插槽，为了说明利用扩展槽进行输入和输出的方法，可将基于 PCM3003 立体声编解码器的音频子板（DSK 工具包中不包含该子板）插入 DSK 板上的 80 脚连接头上。这块音频子板可从 TI 公司买到，附录 F 对该子板的特性进行了描述。PCM3003 编解码器具有可变的抽样速率，最高可达 72 kHz，这对于需要较高抽样速率的应用是很有用的，另外它还含有两个可用的输入输出通道。

DSK 板上有 16 MB 的同步动态 RAM（SDRAM）和 128 KB 的 Flash ROM。两个连接头用来作为输入和输出接口，并分别标为 IN（J7）和 OUT（J6）。4 个用户 DIP 开关中，有 3 个开关的状态可以通过程序读取（有一个程序实例利用这 3 个开关实现语音加扰）。板上的 DSP 时钟频率是 150 MHz，另外，DSK 板上的稳压器为 C6711 核提供 1.8 V 的电源电压，为存储器和外围电路提供 3.3 V 的电源电压。

1.2.2 TMS320C6711 数字信号处理器

TMS320C6711 基于超长指令字（VLIW）的体系结构，非常适合于高强度的数学运算。内部程序存储器采用特殊的设计结构，使每个时钟周期可取 8 条指令。例如，假设 DSP 时钟速率为 150 MHz，那么 C6711 能够在 6.66 ns 内取到 8 个 32 位的指令。

C6711 还有 72 KB 的内部存储器、6 个 ALU 单元和两个乘法器单元组成的 8 个功能或执行单元、寻址空间达到 4 GB 的 32 位地址线和两组 32 位的通用寄存器。

C67xx（如 C6701 和 C6711）属于 C6x 浮点处理器系列，而 C62xx 和 C64xx 属于 C6x 定点处理器系列。C6711 既能进行定点处理也能进行浮点处理，关于它的结构和指令集将在第 3 章讨论。

1.3 程序代码编辑调试软件

程序代码编辑调试软件（CCS）是一个集成软件工具的集成开发环境（IDE），它不仅提供产生程序代码的工具，如 C 编译器、汇编器和连接器，还具有绘图功能以及支持实时调试功能。该软件易于使用，是实现产生程序代码和调试程序的软件工具。

C 编译器编译扩展名为.c 的 C 源程序，生成扩展名为.asm 的汇编源文件。汇编器汇编扩展名为.asm 的汇编源文件，生成机器语言的目标文件，其扩展名为.obj。连接器将目标文件和库文件连接起来，生成扩展名为.out 的可执行文件。该可执行文件表示成已连接的通用目标文件格式（COFF），这种格式在 UNIX 操作系统中很流行，也被许多数字信号处理器的制造商采用[21]。可执行程序可以加载到系统上，直接在 C6711 处理器上运行。

为了创建一个应用工程，可以使用“add”菜单，在该工程中添加合适的文件。可以简单设置

① 1 英寸 = 2.54 cm ——编者注。

编译器/连接器的选项，并可以利用多种调试功能，包括设置断点、观察变量、存储器和寄存器内容以及 C 和汇编的混合编程程序代码、图形化结果、监控执行时间等，还可用不同方法执行程序（如跟踪进入函数、跳过函数、跳出函数等方式）。

可以通过和 DSP/BIOS（参见附录 G）相关的实时数据交换（RTDX）实现数据的实时分析，RTDX 可将主机和目标板实现实时数据交换，不需要停止目标板来进行实时分析，并能实时监控关键的统计数据和性能。通过 JTAG 接口和片上的仿真通信工具通信，从而控制和监控程序的运行。C6711 DSK 目标板上有一个 JTAG 仿真接口。

1.3.1 CCS 的安装和支持

首先用（打印机的）并行电缆 DB25 将 DSK 目标板（J2）和 PC 的并行接口（LP1 或 LP2）连接起来，然后将 DSK 工具包中的 5 V 电源适配器连接到电源接头 J4 上，再接通 DSK 电源开关，用 DSK 附带的 CD-ROM 安装 CCS，安装中最好采用 c:\ti 路径（默认值）。

计算机桌面上的 CCS 图标是“CCS 2['C 6000]”，可双击它来运行 CCS，使用的程序代码生成工具（C 编译、汇编器、连接器）的版本是 4.1。

在通电时，在 4 个用户 DIP 开关附近的 3 个发光二极管（LED）灯会从 1 到 7 进行计数（二进制）。

CCS 提供了有关下面内容的文档，这些文档包含在 DSK 工具包中（参见 Help 图标）:

1. 程序代码生成工具（编译器、汇编器、连接器等）。
2. CCS、编译器、RTDX 以及高级 DSP/BIOS 手册。
3. DSP 指令和寄存器。
4. RTDX、DSP/BIOS 工具等。

CCS[22-34]中包含许多支持材料（PDF 文件），随 CCS 还附带有许多程序实例，如 DSK 性能测试程序、音频程序以及与板上 Flash 有关的程序实例。

本书中的程序是用第 2 版 CCS 来生成和测试的，c:\ti 目录里包含下面一些子文件夹和目录，它们中的许多文件是非常有用的：

1. docs：包含文档和用户手册。
2. myprojects：工程文件夹，本书中讨论的所有程序和工程都可保存在该子目录下面。
3. c6000\cgtools：包含程序代码生成工具。
4. bin：包含许多实用程序。
5. c6000\examples：包含 CCS 中的编程实例。
6. c6000\rtdx：包含实时数据交换的支持文件。
7. c6000\bios：包含 DSP/BIOS 的支持文件。

1.3.2 有用的文件类型

不同的文件有不同的扩展名，比如下面的文件扩展名：

1. .pjt：创建或建立的工程文件。
2. .c：C 源程序文件。
3. .asm：程序员、C 编译器或线性汇编优化器创建的汇编源程序文件。

4. .sa：线性汇编源程序文件，线性优化器输入.sa文件，生成汇编源程序.asm文件。
5. .h：头文件。
6. .lib：库文件，如实时运行支持库文件rts6701.lib。
7. .cmd：连接命令文件，这种文件将段映射到存储器。
8. .obj：汇编器创建的目标文件。
9. .out：连接器创建的可执行文件，这种文件可加载到处理器上并可运行。

1.4 测试DSK工具的编程实例

本节用三个编程实例来说明CCS和DSK电路板的特点，主要任务是熟悉软件和硬件工具，强烈建议在学习后续章节前，练习完这三个编程例子。

1.4.1 DSK的快速测试

从桌面图标启动CCS，依次选择菜单GEL→Check DSK→Quick Test，菜单Quick Test可用于确定操作和安装是否正确，之后将显示如下信息：

Switches: 7

Revision: 2

Target is OK

显示以上信息的前提是假定前三个开关：USER_SW1、USER_SW2和USER_SW3都在上面位置（也就是ON位置）。再将这三个开关设置为110（以二进制表示），这时前两个开关设置在上面位置（第三个开关设置在下面位置），第四个开关是不用的。

重复上面的步骤，依次选择菜单GEL→Check DSK→Quick Test，这时开关的值显示为3（显示信息为"Switches: 3"）。可将开关值设置为从0到7的任何一个数。在程序设计中，可根据这8个不同的值来引导程序的执行。注意Quick Test使发光二极管（LED）周期性闪亮三次。

包含在DSK中的性能检测编程实例，可用来检测和检验DSK上的主要器件是否正常工作，如中断、LED、SDRAM、DMA、串行口及定时器等。

DSK的另一种快速测试方法

1. 从桌面图标打开/启动CCS，然后选择菜单File→Load Program，读取从网上下载的辅助材料，点击文件夹sine8_intr，打开（装载）文件sine8_intr.out，这样就将可执行文件sine8_intr.out装载到C6711处理器中。
2. 选择菜单Debug→Run，将DSK板上的OUT（连接头J6）与扬声器或示波器相连，检验是否有1 kHz的音频信号。DSK板上的IN/OUT连接头是一个3.5 mm的音频插座。

文件夹sine8_intr有支持例1.1的必要文件，通过例1.1可以说明这些工具的一些特点。

1.4.2 支持文件

在PC硬盘上创建一个名为sine8_intr的新文件夹，建议将该文件夹放在目录c:\ti\myprojects里（假设CCS的安装目录为c:\ti）。本书例子中使用的一些支持文件在辅助材料的support文件夹里。暂且不必过分担心这些文件的内容或功能，DSK工具包附带的CCS CD-ROM中还有其他一些支持文件，将下面所列的支持文件从文件夹support复制到硬盘上所建的sine8_intr文件夹中。

1. c6xdsk.cmd：连接命令文件样本。
2. c6xdsk.h：定义外部存储器接口、串行口等地址的头文件（CCS 中 TI 的支持文件）。
3. c6xinterrupts.h：包含中断的 init 函数（DSK 中 TI 的支持文件）。
4. c6xdskinit.h：包含函数原型的头文件。
5. c6xdskinit.c：含有 CCS 中程序实例 codec_poll 用到的几个函数，包括 DSK、编解码器、串行口及输入输出的初始化函数。
6. vectors_11.asm：vectors.asm 的版本包含在 CCS 中，但可以修改它来处理中断。可以使用从 INT4 到 INT15 的 12 个中断，该矢量文件中选择中断 INT11。

另外，还要将 C 源文件 sine8_intr.c 和 GEL 文件 amplitude.gel 从辅助材料 sine8_intr 文件夹中复制到硬盘文件夹 sine8_intr 中。

注意，如果使用 C6211 DSK（现已停止生产），则要修改文件 c6xdskinit.c 中的函数 comm_intr，将其中的 XINT0 改为 XINT1，这是由于 C6211 芯片中存在一个程序漏洞（Bug）。

1.4.3 程序实例

例 1.1 用 8 点产生正弦波

本例用查表方法产生一个正弦波。虽然其功能简单，但重要的是，它展示了 CCS 在编辑、创建工程、访问程序代码生成工具和在 C6711 上运行程序的一些特点。图 1.2 是产生正弦信号的 C 源程序 sine8_intr.c。

```
//sine8_intr.c Sine generation using 8 points, f=Fs/(# of points)
//Comm routines and support files included in C6xdskinit.c

short loop = 0;
short sin_table[8] = {0,707,1000,707,0,-707,-1000,-707}; //sine values
short amplitude = 10;                               //gain factor

interrupt void c_int11()                            //interrupt service routine
{
 output_sample(sin_table[loop]*amplitude); //output each sine value
 if (loop < 7) ++loop;                              //increment index loop
 else loop = 0;                                     //reinit index @ end of buffer
 return;                                            //return from interrupt
}

void main()
{
  comm_intr();                                      //init DSK, codec, McBSP
  while(1);                                         //infinite loop
}
```

图 1.2 用 8 点产生正弦波的程序（sine8_intr.c）

程序中考虑的问题

尽管本例的重点是介绍一些工具，但是理解程序 sine8_intr.c 是很有帮助的。程序创建表或缓

冲区 sin_table，当 $t = 0, 45, 90, 135, 180, 225, 270$ 和 315 度时，在表中或缓冲区中存放 sin(t)的 8 个点的值（8 个点的值都乘以 1000）。在 main 函数中，调用另一个函数 comm_intr，该函数在通信支持文件 c6xdskinit.c 中，它初始化 DSK、DSK 板上的 AD535 编解码器以及 C6711 处理器中的两个多通道缓冲串行端口（McBSP）。

main 函数中的 while(1)语句产生一个无限循环，等待中断发生。一旦有中断发生，程序就进入中断服务程序（ISR）c_int11 执行，ISR 地址在文件 vectors_11.asm 中由分支指令指定，这里使用中断 INT11，第 3 章中将详细讨论中断问题。

在 ISR 中，调用通信支持文件 c6xdskinit.c 中的 output_sample 函数，用它来输出表或缓冲区中的第一个数据 sin_table[0] = 0。选择数据的循环变量依次增加直到指向表的结尾，这时，该循环变量重新初始化为 0，程序执行由 ISR 返回到无限循环 while(1)，等待下一次中断的发生。

每个抽样周期为 $T = 1/F_s = 1/8000 \approx 0.125$ ms，每隔这个时间段产生一次中断。每次中断发生后，就进入中断服务程序，表 sin_table（amplitude = 10 定标）中的一个个数据依次送到编解码器的输出上，从而产生正弦信号。输出信号的周期为 $T = 8 \times 0.125 = 1$ ms，相应的频率为 $f = 1/T = 1$ kHz。

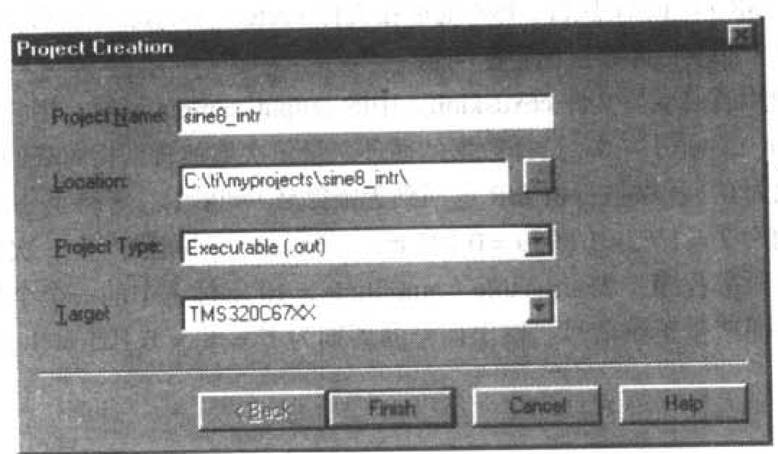
创建工程

在本节中，主要介绍如何创建一个工程。我们考虑为了创建名为 sine8_intr 的工程，需要添加哪些必要的文件。点击桌面图标，进入 CCS 运行环境。

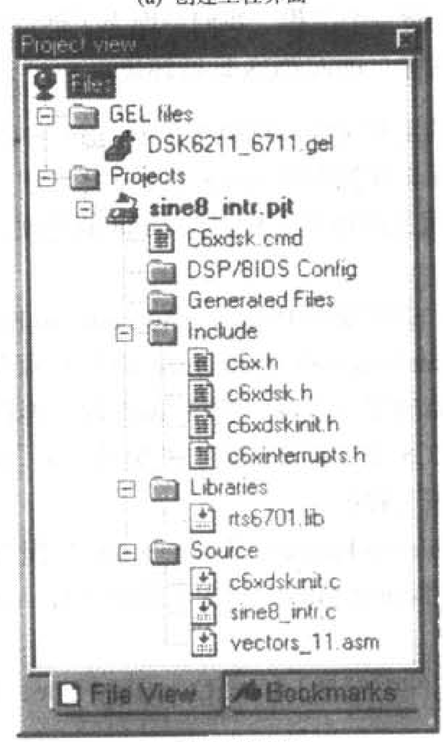
1. 创建名为 sine8_intr.pjt 的工程文件：选择菜单 Project→New，键入工程文件名 sine8_intr，如图 1.3(a)所示，该工程文件保存在 sine8_intr 文件夹中（它位于 c:\ti\myprojects 文件夹下）。扩展名为.pjt 的工程文件保存工程文件 build 的选项、源文件名和相互关系等有关该工程的信息。
2. 在工程文件中添加文件：选择菜单 Project→Add Files to Project，在 sine8_intr 工程中查看 C 源程序文件，打开 c6xdskinit.c 和 sine8_intr.c 两个 C 源文件。一次打开一个文件并添加到工程中；或者将鼠标指向某个文件，按住 Shift 键，再将鼠标指向另一个文件，然后点击 Open 图标，再点击 CCS 中工程文件窗口左边的“+”符号。将工程目录展开后，检查 C 源文件是否已添加到该工程中。
3. 选择菜单 Project→Add Files to Project，在 sine8_intr 工程中查找文件，使用选择文件类型的下拉菜单，选择 ASM Source Files，双击汇编源文件 vectors_11.asm，将它打开/添加到工程中。
4. 重复步骤 3，但选择的文件类型是 Linker Command File，将连接命令文件 c6xdsk.cmd 添加到工程中。
5. 重复步骤 3，但选择的文件类型是 Object and Library Files，查看目录 c\:ti\ c6000 \cgtools\lib，选择实时运行支持库文件 rts6701.lib（它支持 C67x/C62x 结构），将它添加到工程中。这里假定是把 CCS 安装在默认目录 c:\ti 下。
6. 检查连接命令文件（.cmd）、工程文件（.pjt）、库文件（.lib）、两个 C 源文件（.c）和汇编文件（.asm）是否已添加到工程中。当创建工程时，GEL 文件 dsk6211_6711.gel 会自动添加到工程中，它会初始化 DSK。
7. 注意这时还没有 include 文件，选择菜单 Project→Scan All Dependencies，这样就可添加头

文件 c6xdsk.h，c6xdskinit.h，c6xinterrupts.h 和 c6x.h。前三个文件可从辅助材料中复制进来，而头文件 c6x.h 包含在 CCS 中。

CCS 中的文件窗口看上去和图 1.3(b)所示类似。除了库文件外，点击 CCS 窗口中任何文件都可以显示它们的内容。当选择 Scan All Dependencies 时，会自动添加头文件或包含文件，而不需要手动添加。



(a) 创建工程界面



(b) 工程文件列表

图 1.3 CCS 创建和显示 sine8_intr 工程时的界面

当然也可以使用简单方法，就是将不同的窗口文件“拖到”CCS 工程窗口中。

代码生成和选项设置

程序代码生成工具 C 编译器和连接器有许多不同的设置选项，为了创建一个工程，需要对这些选项进行不同的设置。

编译器选项

选择菜单 Project→Build Options，图 1.4(a)显示了编译器在 Build Options 选项时的 CCS 窗口。选择下面的编译器选项：(a) Basic（在 Category 选项栏中选择）；(b) Default（在 Target Version 选项栏中选择）；(c) Full Symbolic Debug（在 Generate Debug Info 选项栏中选择）；(d) Speed most critical（在 Opt Speed vs. size 选项栏中选择）；(e) None（在 Opt Level 和 Program Level Opt 选项栏中选择）。结果编译器选项变成：

```
-gks
```

–k 选项表示保留汇编源文件 sine8_intr.asm。–g 选项允许显示符号调试信息，这在调试过程是非常有用的，将它与–s 选项一起使用可将 C 源文件和产生的汇编文件 sine8_intr.asm 交叉列出清单来。–g 选项表示禁止程序代码优化，以便使调试过程更加方便。

在 Target Version 选项栏选择 Default，表示启用定点实现方式（如果使用 C6211 DSK，则必须使用该选项）。C6711 DSK 既可以使用定点处理方式，也可以使用浮点处理方式，本书中大多数程序例子可用定点方式运行。对于第 6 章和第 7 章中的程序例子，就要选择 C671x，启用浮点处理方式。

如果选择 No Debug（在 Generate Debug Info 选项栏选择）以及–o3: File（在 Opt Level 选项栏中选择），编译器选项就会自动改成：

```
-ks -o3
```

–o3 选项表示对性能和执行速度进行最高级的优化，因为暂时对速度要求并不是很严格（调试时也是这样）。编译器选项-gks 也可以在编译命令窗口中直接键入。为了调试方便，开始调试时，不需要对速度进行优化，文献[26]中对许多编译器选项进行了介绍。

连接器选项

从 CCS Build Options 窗口中点击 Linker 选项，在 Output Module 选项栏中选择 Absolute Executable，在 Output Filename 选项栏中键入 sine8_intr.out，在 Autoinit Model 选项栏中选择 Run-time Autoinitialization，输出文件名默认是.pjt 文件，这时连接器选项应如图 1.4(b)所示，也就是：

```
-g -c -o "sine8_intr.out" -x
```

–c 选项用于在运行时初始化变量，–o 选项用于命名连接后输出的可执行文件名 sine8_intr.out。按下 OK 按钮。

注意，可以选择把可执行文件保存到一个子文件夹 Debug 中，特别是在调试一个工程期间，这样做是非常必要的。

同样，也可以直接从适当的窗口键入这些命令。

建立和运行工程

现在可建立和运行工程 sine8_intr，过程如下所示。

1. 建立工程 sine8_intr：选择菜单 Project→Rebuild All，或按下带有三个向下箭头的工具条，这样 CCS 就会调用 cl6x 对所有 C 源文件进行汇编和编译，调用 asm6x 对汇编文件 vectors_11.asm 进行编译，lnk6x 将生成的目标文件和运行库支持文件 rts6701.lib 连接起来，最后生成可执行文件 sine8_intr.out，该文件可直接加载到 C6711 处理器上运行。注意汇编、

编译和连接的命令要和 Build options 中的设置配合在一起执行。上述命令执行后，还会生成一个记录文件 cc_build_debug.log，文件中包含经过汇编、编译的文件以及所选择的编译选项设置的信息，同时还列出了使用的支持函数。图 1.5 显示了 CCS 工程 sine8_intr 的几个窗口。

2. 选择菜单 File→Load Program，通过点击 sine_intr.out 装载该文件（CCS 有一个选项，在建立工程后会自动装载该程序），它应该在 sine8_intr 文件夹中。选择菜单 Debug→Run，或使用工具栏中的“跑步者”图标，将 DSK 板上的 OUT 连接头（J6）和扬声器连接起来，就可听到音频信号。

(a) 编译器子窗口

(b) 连接器子窗口

图 1.4　CCS 选择 Build Options 时的窗口

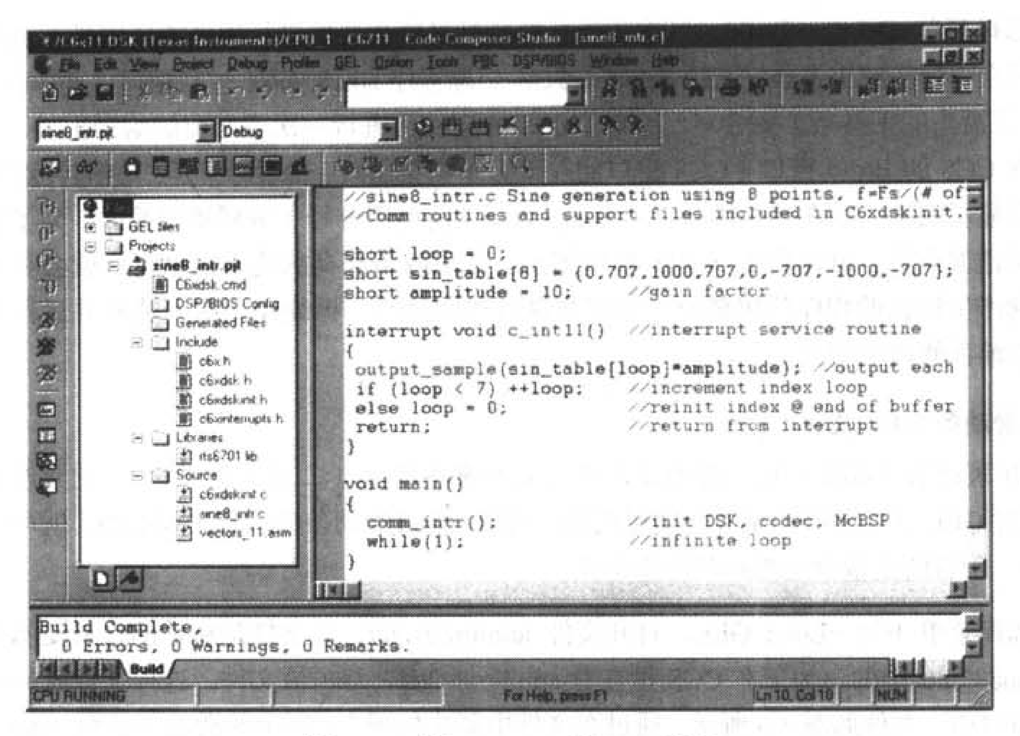


图 1.5 工程 sine8_intr 的 CCS 窗口

编解码器抽样频率 F_s 固定在 8 kHz，产生的频率 $f = F_s$/点数 = 8 kHz/8 = 1 kHz，将 DSK 的输出连接到示波器上，检验是否有振幅（峰-峰值）大约为 0.85 V、频率为 1 kHz 的正弦信号。

Watch Window（观察窗口）

检查处理器是否仍在运行，注意在 CCS 的左下方有一个“DSP RUNNING”指示标志。从观察窗口可以改变参数的值，或者监视变量随程序运行的变化情况，过程如下所示。

1. 选择菜单 View→Quick Watch window，观察窗口就出现在 CCS 的下方，输入 amplitude 变量，然后点击 Add to Watch，程序中设置幅度为 10，该值就会显示在观察窗口里，如图 1.2 所示。
2. 将参数 amplitude 的值从 10 改成 30。
3. 检查产生的音频音的音量是否已经增大了（注意处理器还在运行），这时正弦波的幅度从大约 0.85 V 增加到大约 2.6 V（峰-峰值）。
4. 将参数 amplitude 的值改为 33（和步骤 2 相同），检验是否音调已变高，这看上去好像只要改变正弦波的幅度就可改变正弦波的频率，事实并非如此，这是因为输出已经溢出了 16 位编解码器 AD535 的范围。由于表中的所有数都再乘以 33，所以这时表中数值的范围在+33000 和−33 000 之间；又因为 AD535 的输出范围限制在-2^{15}到（$2^{15}-1$）之间，即从−32 768 到+32 767，所以导致输出产生溢出现象。不要试着将超过 16 位的数据送到编解码器，否则可能导致溢出。板上的编解码器数据格式采用的是二进制补码。

纠正程序错误

1. 删除语句 `short amplitude = 10;` 中的分号。如果没有显示 C 源文件 sine8_intr，则从文件窗口双击该文件。

2. 选择菜单 Debug→Build，该选项执行的是只对单个文件进行编译或使用带有两个（不是三个）箭头的工具条，选择 Build 是只对 C 源程序 sine8_intr.c 进行编译，而 Rebuild 选项（三箭头的工具条）则会对已编译或汇编的文件再进行一次不必要的编译汇编。
3. 在 CCS 的 Build 窗口（CCS 窗口的左下方）就会出现一条错误消息，并且用红色的高亮行显示出来，并在后面用 *a ";" is expected*（需要用分号）解释错误原因。为了更好地显示错误信息，就需要将 Build 窗口缩小。如果双击高亮的错误信息行，就会将光标移动到产生错误的程序代码部分。对程序做适当的修改，再编译一次，装载并运行程序，检验前面修改的结果。

使用滑动条 GEL 文件

通用扩展语言（GEL）是一种类似于 C 语言的解释语言（C 语言的子集），它允许处理器在运行时通过 GEL 文件的滑动条改变变量的值。例如，通过滑动条滑动到不同的值，就可以改变信号的幅度，所有的变量事先要在程序中定义。

1. 选择菜单 File→Load GEL，打开文件 amplitude.gel，该文件是从辅助材料复制到文件夹 sine8_intr 中的。为了在 CCS 里查看 amplitude.gel，双击该文件，这样文件就显示在文件窗口中，文件如图 1.6 所示。通过在文件中创建如图 1.6 所示的滑动条函数 amplitude，开始时选择幅度变量的初始值 10（第一个值），它是在 C 程序中设置的，最高值为 35（第二个值），按照步进值为 5 往上递增（第三个值）。

```
/*Amplitude.gel Create slider and vary amplitude of sinewave*/

menuitem "Sine Amplitude"

slider Amplitude(10,35,5,1,amplitudeparameter)  /*start at 10,up to 35*/
      {
        amplitude = amplitudeparameter;           /*vary amplit of sine*/
      }
```

图 1.6 在正弦信号产生程序中，用滑动条选择不同幅度的 GEL 文件（amplitude.gel）

2. 选择菜单 GEL→Sine Amplitude→Amplitude，这时将会弹出如图 1.7 所示的滑动条窗口，幅度的最小值为 10。
3. 按一下键盘的向上箭头键，如滑动条窗口显示那样，将幅度值从 10 增加到 15，检验产生的正弦波的音量是否增大了。再按一下向上的箭头键，继续提高滑动条指针位置，幅度由 5 增加到 30。当 amplitude 设为 30 时，正弦波的幅度应该是 2.6 V 左右（峰-峰值）。现在，用鼠标点击滑动窗口，慢慢地将滑动条指针的位置提高到 31，然后再提高到 32，检验产生的信号频率是否还是 1 kHz。当滑动条指针增加到 33 时，检查所产生的信号不再是 1 kHz 正弦波（而是含有 1 kHz 和 3 kHz 两个正弦音的信号）。表中的所有数和幅度变量的值相乘，从而使输出在+33 000 到−33 000 之间（超过了编解码器可以接受的范围）。

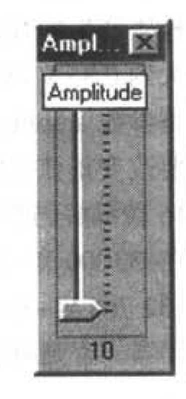


图 1.7 改变正弦波幅度的 CCS 滑动条窗口

和前面使用一个滑动条一样，也可以很方便地使用两个滑动条：一个控制幅度，另一个控制频率。在C程序中，通过改变循环变量来产生不同的频率（参见例2.4，在表中每两点步进一次）。当创建完一个工程后，关闭CCS时，有关该工程所有改变的选项就会保存下来。当再次打开该工程时，其设置的状态和上次关闭该工程时是一样的。

例1.2 产生正弦信号和使用CCS画图

和例1.1一样，本例也产生8个点的正弦信号，更重要的是，它介绍了CCS在频域和时域中的画图功能。程序sine8_buf.c（如图1.8所示）实现了该工程，它在存储器中创建一个缓冲区，并将输出数据存储到该缓冲区中。

```
//sine8_buf Sine generation. Output buffer plotted within CCS
//Comm routines and support files included in C6xdskinit.c

short loop = 0;
short sine_table[8] = {0,707,1000,707,0,-707,-1000,-707};  //sine values
short out_buffer[256];                   //output buffer
const short BUFFERLENGTH = 256;          //size of output buffer
short i = 0;                             //for buffer count

interrupt void c_int11()                 //interrupt service routine
{
  output_sample(sine_table[loop]);       //output each sine value
  out_buffer[i] = sine_table[loop];      //output to buffer
  i++;                                   //increment buffer count
  if (i == BUFFERLENGTH) i = 0;          //if bottom reinit buffer count
  if (loop < 7) ++loop;                  //increment index loop
  else loop = 0;                         //if end of buffer,reinit index
  return;
}

void main()
{
  comm_intr();                           //init DSK, codec, McBSP
  while(1);                              //infinite loop
}
```

图1.8 正弦波产生程序，输出保存在存储器中（sine8_buf.c）

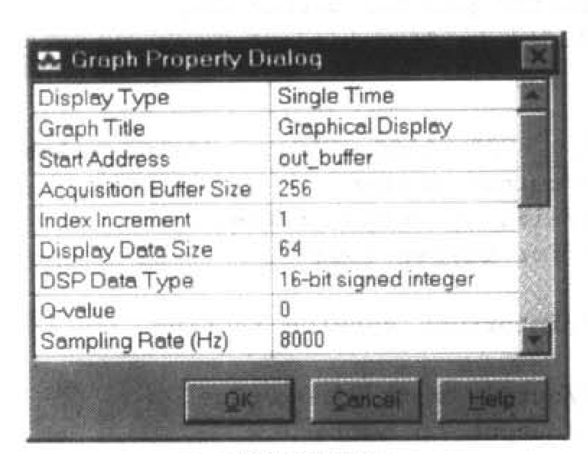
创建名为sine8_buf.pjt的工程，和例1.1一样，在工程中添加必要的文件（用sine8_buf.c代替sine8_intr.c）。注意，要选择菜单Project→Scanning All Dependencies，将必要的支持头文件添加到该工程中，该工程的所有支持文件在sine8_buf文件夹里（在辅助材料中）。

建立工程sine8_buf，装载并运行可执行文件sine8_buf.out，通过连接到输出的扬声器或示波器，检查是否产生了1 kHz的正弦信号（和例1.1一样）。

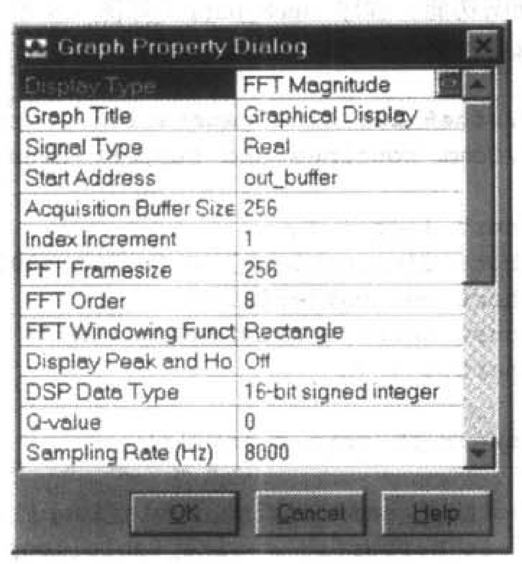
用CCS画图

输出缓冲区不断地更新，每隔256个点更新一次（改变缓冲区大小非常容易），下面介绍用CCS画出存储在缓冲区out_buffer中当前输出数据的步骤。

1. 选择菜单 View→Graph→Time/Frequency。
2. 改变 Graph Property Dialog 中的选项，如图 1.9(a)所示，将选项设置为时域图（适当使用下拉菜单），并将输出缓冲区的起始地址设置为 out_buffer，其他选项保留默认值，图 1.10 显示了正弦信号的时域图。
3. 图 1.9 (b)给出了 CCS 用 Graph Property Display 显示的频域图，选择 FFT 的阶数 n，以便用 2^n 作为 FFT 的帧长。按下 OK 按钮，检验 FFT 的幅频图是否和图 1.10 一样，1000 Hz 处的尖峰信号表示产生的正弦信号。



(a) 时域图特性对话框



(b) 频域图对话框

图 1.9 工程 sine8_buf 在 CCS 中 Graph Property Dialog 窗口的选项设置

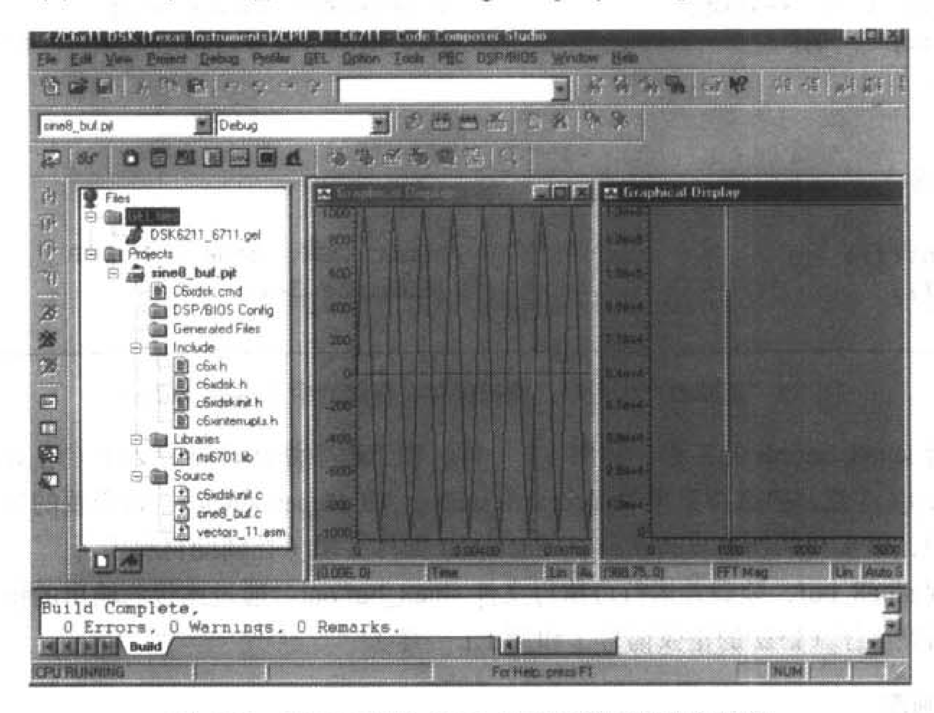


图 1.10 CCS 画出的 1 kHz 正弦波的时域和频域图

注意，为改变屏幕大小，在 Build 窗口点击右键，取消 Allow Docking 的选项，就可在 CCS 中得到很多不同的窗口。

例 1.3 两数组的点积

在数字信号处理器中，像加/减和乘法运算都是关键运算，乘法/累加运算更加重要，它在许多需要数字滤波、相关和频谱分析的应用中相当有用。乘法运算非常普遍，而且是大多数字信号处理算法的基础，因此在一个时钟周期内执行乘法运算是很重要的。使用 C6x，实际上可以在一个时钟周期内执行两次乘法/累加运算。

本例介绍 CCS 的其他特点，例如单步执行和对程序进行剖面分析，测试各种性能，重点仍然是熟悉使用 CCS 工具。调用 C 编译优化器后，看看性能和执行速度怎样才能显著提高。

C 源文件 dotp4.c（如图 1.11 所示）实现两个数组的点积，每个数组有 4 个数，头文件 dotp4.h 定义了这两个数组（见图 1.12）。第一个数组包含 1, 2, 3, 4 四个数，第二个数组包含 0, 2, 4, 6 四个数，两个数组的点积是$(1\times 0)+(2\times 2)+(3\times 4)+(4\times 6)=40$。

```
//Dotp4.c Multiplies two arrays, each array with 4 numbers

int dotp(short *a, short *b, int ncount);      //function prototype
#include <stdio.h>                             //for printf
#include "dotp4.h"                             //data file of numbers
#define count 4                                //# of data in each array
short x[count] = {x_array};                    //declara 1st array
short y[count] = {y_array};                    //declara 2nd array

main()
{
 int result = 0;                               //result sum of products

 result = dotp(x,y,count);                     //call dotp function
 printf("result = %d (decimal) \n", result); //print result
}

int dotp(short *a, short *b, int ncount)       //dot product function
{
 int sum = 0;                                  //init sum
 int i;

 for (i = 0; i < ncount; i++)
   sum += a[i] * b[i];                         //sum of products
 return(sum);                                  //return sum as result
}
```

图 1.11 求两个数组点积的 C 程序

```
//dotp4.h Header file with two arrays of numbers

#define x_array 1,2,3,4

#define y_array 0,2,4,6
```

图 1.12 定义两个数组且每个序列有 4 个数的头文件

这个程序可以很方便地修改，以处理更大的数组，因为本例没有使用实时实现，因此不需要实时 I/O 支持文件。这里也不需要中断支持文件，用到的矢量文件也不大，如图 1.13 所示。

```
*Vectors.asm Vector file for non-interrupt driven program
     .title        "vectors.asm"
     .ref          _c_int00            ;reference entry address
     .sect         "vectors"           ;in vector section
rst: mvkl    .s2   _c_int00,b0         ;lower 16 bits -> b0
     mvkh    .s2   _c_int00,b0         ;higher 16 bits -> b0
     b       .s2   b0                  ;branch to entry address
     nop                               ;5 NOPs for rest of fetch packet
     nop
     nop
     nop
     nop
```

图 1.13 非中断驱动程序的矢量文件（vectors.asm）

创建和建立工程 dotp4，和例 1.1 一样，将下面的文件添加到工程中：

1. C 源文件 dotp4.c。
2. 定义入口地址 c_int00 的矢量文件 vectors.asm。
3. 连接命令文件 c6xdsk.cmd。
4. 库文件 rts6701.lib。

因为可以选择菜单 Project→Scan All Dependencies 将所需要的 include 文件添加到工程的文件夹中，因此不要使用菜单 Add Files to Project。程序 dotp4.c 中使用了 printf 语句打印结果，因此需要头文件 stdio.h。

在观察窗口设置观察变量

1. 选择菜单 Project→Options，进行如下设置：
   编译器选项设置为–gs；
   连接器选项设置为–c –o dotp4.out。
2. 选择三箭头的工具条，执行 Rebuild All 命令（或选择菜单 Debug→Build）。
3. 选择菜单 View→Quick Watch，输入观察变量 sum，点击 Add to Watch，这时会显示与 sum 相关的一条消息 "identifier not found"，这是因为现在仍在 main 函数中，局部变量还不存在。
4. 为了在程序行：

   ```
   sum += a[i] * b[i];
   ```

   设置断点，将鼠标放在该行（点击），然后右击鼠标选择 Toggle breakpoint，这时程序行的左边应该出现一个圆圈。
5. 选择菜单 Debug→Run（或使用"跑步者"工具条），程序将执行到设置断点的行，这时一个黄色箭头就会指向这行程序。
6. 使用 F8 键进行单步执行（或使用工具条），继续单步执行，在观察窗口观察变量 sum 值从 0, 4, 16 变化到 40。选择菜单 Debug→Run，检查打印的 sum 结果是否为：

   sum = 40（十进制）

7. 注意，C 程序 dotp4.c 中使用了打印结果的 printf 语句，应该避免使用这种语句，因为执行这条语句需要 3000 个时钟周期。

仿真

1. 选择菜单 Debug→Reset CPU→File→Reload Program，重载执行文件 dotp4.out。
2. 和前面一样，再次在相同的程序行设置断点，选择菜单 Debug→Animate，通过观察窗口观察变量 sum 值的变化。另外，通过选择菜单 Option→Customize→Animate Speed，可控制仿真的速度。

测试没有经过优化的程序性能（剖面分析）

本节介绍如何将一段程序代码作为测量基准，例中仍使用 dotp 函数，检查编译器、连接器选项的设置是否和以前设置的一样，即编译器选项设置为–gs，连接器选项设置为–c –o dotp4.out。

为了分析程序各个方面的性能以及产生符号调试信息，编译器选项必须使用–g。为了去掉程序行中的断点，可右击设置断点的程序行，再选择菜单 Toggle breakpoint，这样就可去掉程序行中的断点。

1. 选择菜单 Debug→Reset CPU→File→Reload Program，重载可执行文件。
2. 选择菜单 Profiler→Start New Session，在 Profile Session Name 中输入 dotp4，然后按下 OK 按钮。
3. 点击 Create Profile Area 图标，即图 1.14(b)左上起第四个图标，图 1.14(b)显示了在 C 源文件 dotp4.c 中添加函数 dotp 的剖面分析区。
4. 运行程序，检查如图 1.14(b)所示的运行结果，结果显示执行函数 dotp 需要 138 个时钟周期（程序没有优化时）。

测试经过优化的程序性能（剖面分析）

本节介绍如何用一种优化选项–o3 来优化程序，通过优化 C 编译器可提高程序的运行速度。选择菜单 Project→Build Options，将编译器选项设置改为：

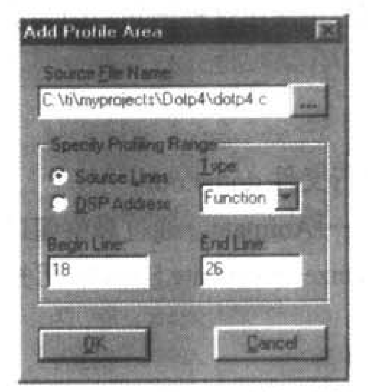
```
-g  -o3
```

连接器选项和以前设置的一样（也可直接输入）。–o3 选项调用最高的编译优化方法，文献[26]对多种编译器选项进行了描述。使用 Rebuild All 功能（带三个箭头的工具条），装载可执行文件 dotp4.out（选择菜单 File→Load Program）。注意装载可执行文件后，执行的入口地址是 c_int00，可由反汇编文件查看入口地址。

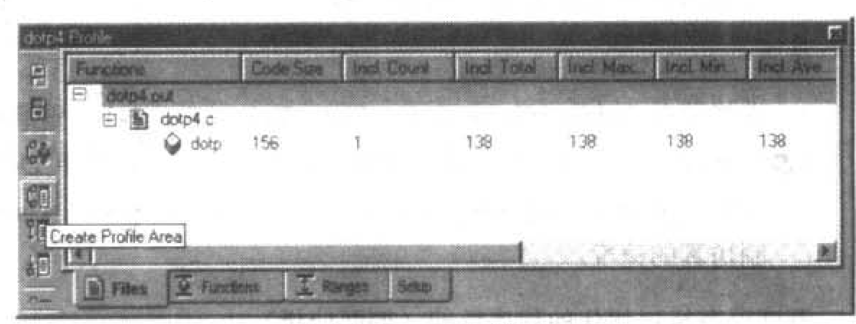
选择菜单 Debug→Run，检查执行 dotp 函数需要的时间，这时发现时间从 138 个时钟周期变成了 30 个时钟周期，如图 1.14(c)所示。可见利用 C 编译器优化方法，程序执行速度有显著提高。还可以利用第 3 章的内部函数和第 8 章的程序代码优化方法对该点积例程进行进一步优化。

1.5 支持程序/文件的一些考虑

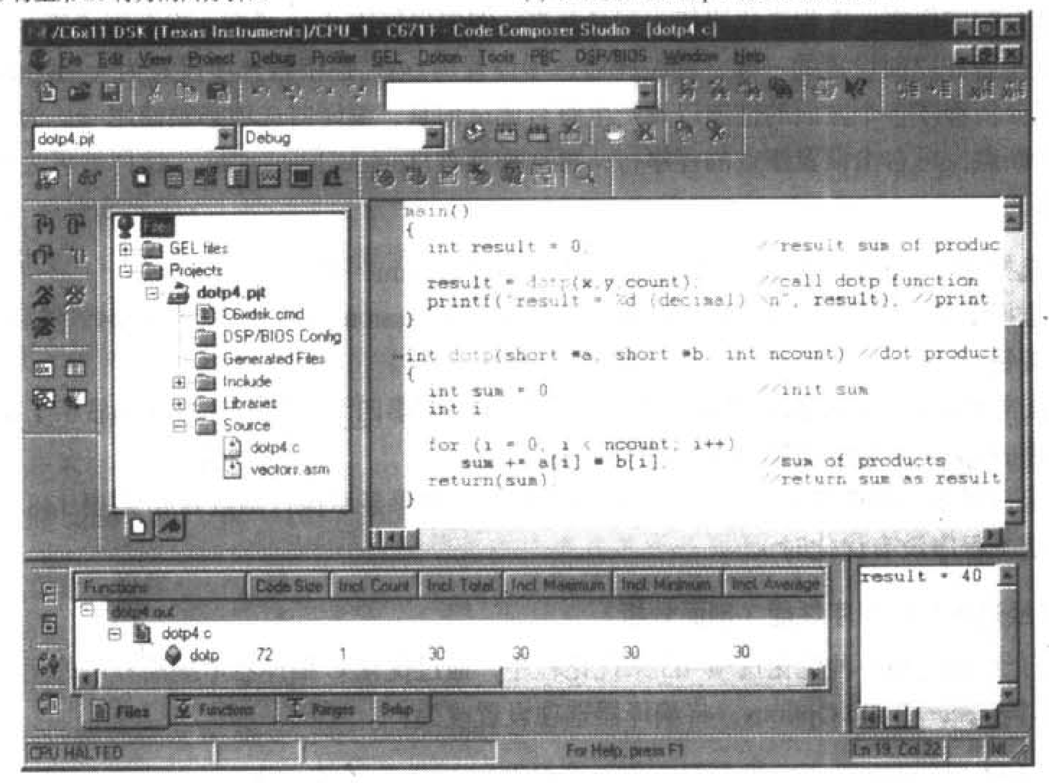
下面的支持文件实际上可用于本书所有的例子中：(1) c6xdskinit.c；(2) vectors_11.asm；(3) c6xdsk.cmd。目前的重点是使用它们。



(a) 设置第 18 行至第 26 行为剖面分析区



(b) 没有优化的 dotp 函数剖面分析窗口



(c) 优化的 dotp 函数剖面分析窗口

图 1.14 工程 dotp4 剖面分析的 CCS 窗口显示

1.5.1 初始化通信文件

C 源程序主程序中的 comm_intr 函数位于通信文件 c6xdskinit.c 中，程序的部分内容如图 1.15 所示。DSK 被初始化，然后配置、启用传输中断 INT11。

该通信支持文件同时包含输入和输出两个函数，函数 input_sample 返回来自 mcbsp0_read 输入的数据，而函数 output_sample 调用 mcbsp0_write 进行输出。

中断驱动程序

使用中断驱动程序，就需要选择一个中断（这里选择 INT11），非屏蔽中断位和全局中断使能位（GIE）一样，都要处于使能（Enable）状态，中断支持函数在支持文件 c6xdskinterrupts.h 中，它们被文件 c6xdskinit.c 中的函数 comm_intr 所调用。

```
//C6xdskinit.c Partial listing. Init DSK,AD535,McBSP

#include <c6x.h>
#include "c6xdsk.h"
#include "c6xdskinit.h"
#include "c6xinterrupts.h"

void mcbsp0_write(int out_data)        //function for writing
{
 int temp;

 if (polling)                           //bypass if interrupt-driven
 {
  temp = *(unsigned volatile int *)McBSP0_SPCR & 0x20000;
  while ( temp == 0)
     temp = *(unsigned volatile int *)McBSP0_SPCR & 0x20000;
 }
 *(unsigned volatile int *)McBSP0_DXR = out_data;
}

int mcbsp0_read()                       //function for reading
{
 int temp;

 if (polling)                           //bypass if interrupt-driven
 {
  temp = *(unsigned volatile int *)McBSP0_SPCR & 0x2;
  while ( temp == 0)
     temp = *(unsigned volatile int *)McBSP0_SPCR & 0x2;
 }
 temp = *(unsigned volatile int *)McBSP0_DRR;
 return temp;
}
void comm_poll()                        //communication with polling
{
   polling = 1;                         //setup for polling
   c6x_dsk_init();                      //call init DSK function
}

void comm_intr()                        //communication with interrupt
{
  polling = 0;                          //if interrupt-driven
  c6x_dsk_init();                       //call init DSK function
  config_Interrupt_Selector(11,XINT0);  //using transmit interrupt INT11
  enableSpecificINT(11);                //for specific interrupt
  enableNMI();                          //enable NMI
  enableGlobalINT();                    //enable GIE global interrupt
  mcbsp0_write(0);                      //write to SP0
}

void output_sample(int out_data)        //added function for output
{
  mcbsp0_write(out_data & 0xfffe);      //mask out LSB
}

int input_sample()                      //added function for input
{
   return mcbsp0_read();                //read from McBSP0
}
```

图 1.15 通信支持程序的部分清单（c6xdskinit.c）

轮询型程序

轮询程序（非中断驱动程序）不断查询或检测发送或接收数据是否已准备好，这种方式没有中断型方式效率高。在输入读函数 mcbsp0_read 中，将串行口控制寄存器（SPCR）的内容和 0x2 进行 AND 操作，用来检测寄存器 b1 位的状态（第二个最低有效位），如图 B.8 所示（见附录 B）。在输出写函数 mcbsp0_write 中，将 SPCR 和 0x20000 进行 AND 操作，用来检测 b17 位的状态。输入数据通过多通道缓冲串行口（McBSP）的数据接收寄存器来接收，输出数据通过 McBSP 的数据发送寄存器来发送。

后面的一些例子采用轮询方式来控制输入和输出数据速度，但大多数例子是中断驱动型的。关于中断将在第 3 章进行讨论，目前，INT11 是由串行口（McBSP）产生的。

1.5.2 矢量文件

中断服务程序(ISR)c_int11 位于 C 程序(sine8_intr.c 或 sine8_buf.c)中，为了选择中断 INT11，指向中断服务程序的分支指令应该放在 vectors_11.asm 文件中地址 INT11 的下面。文件 vectors_11.asm 如图 1.16 所示，注意调用的程序或函数名前有下划线，在 vectors_11.asm 中的 ISR 也用.ref_c_int11 来表示。

对于非中断驱动型矢量文件，按照下面的方法修改文件 vectors_11.asm：

1. 删除中断服务程序.ref_c_int11 的交叉引用参考语句.ref _c_int00。
2. 对于中断 INT11，用 NOP 指令替代指向 ISR 的分支指令。

```
*Vectors_11.asm Vector file for interrupt-driven program
            .ref          _c_int11      ;ISR used in C program
            .ref          _c_int00      ;entry address
            .sect         "vectors"     ;section for vectors
RESET_RST:  mvkl    .S2   _c_int00,B0   ;lower 16 bits -> B0
            Mvkh    .S2   _c_int00,B0   ;upper 16 bits -> B0
            B       .S2   B0            ;branch to entry address
            NOP                         ;NOPs for remainder of FP
            NOP                         ;to fill 0x20 Bytes
            NOP
            NOP
            NOP
NMI_RST:    .loop 8
            NOP                         ;fill with 8 NOPs
            .endloop
RESV1:      .loop 8
            NOP
            .endloop
RESV2:      .loop 8
            NOP
            .endloop
INT4:       .loop 8
            NOP
            .endloop
INT5:       .loop 8
            NOP
            .endloop
INT6:       .loop 8
            NOP
            .endloop
```

```
INT7:           .loop 8
                NOP
                .endloop
INT8:           .loop 8
                NOP
                .endloop
INT9:           .loop 8
                NOP
                .endloop
INT10:          .loop 8
                NOP
                .endloop
INT11:          b       _c_int11            ;branch to ISR
                .loop 7
                NOP
                .endloop
INT12:          .loop 8
                NOP
                .endloop
INT13:          .loop 8
                NOP
                .endloop
INT14:          .loop 8
                NOP
                .endloop
INT15:          .loop 8
                NOP
                .endloop
```

图 1.16　中断驱动矢量程序（vectors_11.asm）

1.5.3　连接器文件

连接器命令文件 c6xdsk.cmd 如图 1.17 所示，它显示了像.text 和.stack 等常驻于 IRAM 的一些段，并给出了映射到 C6711 数字信号处理器内存中的状况。尽管命令文件的某些部分是不必要的，但它可作为通用的连接命令文件模板。第 4 章给出了一个使用 SDRAM 作为外部 RAM 的例子，其起始地址为 0x80000000。

1.6　编译器/汇编器/连接器的 Shell 程序

在前面的例子中，当建立一个工程时，就会调用 CCS 中进行编译、汇编和连接的程序代码生成工具，这些工具也可以在 CCS 外用 DOS shell 程序直接调用。

1.6.1　编译器

编译器 Shell 程序可用下面命令行调用：

```
Cl6x [options] [files]
```

这条命令可对扩展名为.c 的 C 文件、扩展名为.asm 的汇编文件和扩展名为.sa 的线性汇编（第 3 章中将介绍）进行汇编和编译。线性汇编程序文件是 C 和汇编的“中间产物”，它实现了 C 程序的通用性和汇编程序的高效性之间的折中。例如：

```
Cl6x -gks -o3 file1.c, file2, file3.asm, file4.sa
```

```
/*C6xdsk.cmd Generic Linker command file*/

MEMORY
{
  VECS:       org =          0h, len =      0x220 /*vector section*/
  IRAM:       org = 0x00000220, len = 0x0000FDC0 /*internal memory*/
  SDRAM:      org = 0x80000000, len = 0x01000000 /*external memory*/
  FLASH:      org = 0x90000000, len = 0x00020000 /*flash memory*/
}

SECTIONS
{
  vectors    :> VECS
  .text      :> IRAM
  .bss       :> IRAM
  .cinit     :> IRAM
  .stack     :> IRAM
  .sysmem    :> SDRAM
  .const     :> IRAM
  .switch    :> IRAM
  .far       :> SDRAM
  .cio       :> SDRAM
}
```

图 1.17 通用连接器命令文件（c6xdsk.cmd）

上面的命令将调用 C 编译器对 file1 和 file2（默认扩展名为.c）进行编译，并生成汇编文件 file1.asm 和 file2.asm，同时调用汇编器优化程序对 file4.sa 进行优化，生成文件 file4.asm，然后汇编器（用 Shell 命令 c16x 调用）汇编这 4 个汇编源文件并生成目标文件 file1.obj, …, file4.obj。选项设置为-gs，分别用于在调试中获得特定的调试信息和将 C 语句与汇编文件交叉列出来，选项设置-k 用于保留生成的汇编源程序。

有 4 种级别的编译器优化选择方案，-o3 调用最高的优化级别；0 级优化给变量分配寄存器，1 级优化兼有 0 级优化的所有功能，并能去掉局部的公共表达式和没有用的赋值语句；2 级优化兼有所有 1 级优化功能并对循环和滚动（Rolling）进行优化（将在后面讨论）；3 级优化具有所有 2 级优化功能，并去除未调用的函数。另外还有使程序代码最短的编译器优化方案（这种优化方案可能会引起程序代码执行速度下降）。

注意，完全优化方案可能会改变存储器单元的分配，这样可能影响程序的功能。这种情况下，必须声明这些存储器单元是易变的。编译器对易变变量不进行优化且将易变变量分配到未初始化的段中，而不是寄存器中。当存储器访问方式在 C 代码中指明时，可以使用易变变量。

在开始优化时，首先关注的是程序的功能。在开始调试时，不要调用任何（或太高的）优化选项，因为提供的一些附加特定调试信息将加快调试过程，这些附加信息会影响程序的性能。在程序进行优化后，很难对程序进行调试，因为程序行并不是按通常的顺序组织的。在进行编译时，也可以用 C_OPTION 环境变量设置编译器选项。

1.6.2 汇编器

汇编文件 file3.asm 也可以用下面命令进行汇编：

```
asm6x file3.asm
```

经汇编后，生成目标文件 file3.obj。扩展名.asm 是可选的，生成的目标文件必须和实时运行支持库相连接，生成扩展名为.out 的可执行公共目标文件格式（COFF），这种格式文件可直接装载到数字信号处理器中运行。

1.6.3 连接器

连接器可用下面语句进行调用：

```
lnk6x -c prog1.obj -o prog1.out -l rts6701.lib
```

选项-c 通知连接器在运行时，使用 C 环境定义规则对变量进行自动初始化（另一个连接器选项是-cr，表示在装载时对变量进行初始化）。选项-l 表示调用运行支持库文件 rts6701.lib，当连接时，必须使用这些选项(-c/-cr和-l)。目标文件prog1.obj 和库文件进行连接，生成可执行文件 prog1.out。如果不使用-o 选项，就生成名为 a.out（默认文件名）的可执行文件。

连接器也可以用带-z 选项的编译命令来调用：

```
Cl6x -gks -o3 prog1.c prog2.asm -z -o prog.out -m prog.map
-l rts6701.lib
```

使用上述命令，生成可执行文件 prog.out。选择选项-m 会生成映射文件，该文件列出所有段、符号和标号地址，对调试程序非常有用。

连接器选项-heap size 指定用于动态存储分配，并按字节计算堆的大小（默认是 1 kB），选项-stack size 则用于指定 C 系统的堆栈大小（单位是字节），其他连接器选项可参见文献[24]。

连接器使用默认的存储单元分配算法将程序分配到存储器中，它将不同的段放到合适的存储区中，这些存储区中存放程序代码和数据。使用扩展名为.cmd 的连接器命令文件，可以使存储器按照用户的要求进行分配，在命令文件中用 MEMORY 和 SECTIONS 指令说明。连接器指令 MEMORY（大写）定义了存储器模式，并指定多个可用的存储器空间的起始地址和长度。连接指令 SECTIONS（大写）将输出段分配到定义的存储器空间中，并将不同的程序代码段分配到可用存储器空间中。

图 1.17 给出了连接程序命令文件模板文件，该文件几乎可用于本书中的所有例子。我们将使用内部存储器（IRAM）来存储代码和数据段。在第 4 章中，我们将说明用外部存储器（SDRAM）来实现数字滤波器。外部程序存储器的起始地址为 0x80000000，长度为 0x1000000，即 16 MB。闪存（Flash）地址从 0x90000000 开始，长度为 0x20000，即 128 kB。

当使用 C 程序来初始化运行环境时，设置入口为 c_int00，连接程序将自动和 boot.obj 进行连接。当使用连接器选项-c（或-cr）时，连接器将自动定义_c_int00 符号，包含在运行支持库中的函数_c_int00 是 boot.obj 的入口，boot.obj 文件建立堆栈并调用主函数。运行库支持程序 boot.c 对变量进行自动初始化，连接器选项-c 将使用 boot.c 进行初始化过程。注意_c_int00 是在矢量文件 vectors_11.asm 和 vectors.asm 中定义的。

参考文献

注：文献[21-33]已包含在 DSK 工具包中。

1. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, Wiley, New York, 1999.

2. R. Chassaing, *Digital Signal Processing with C and the TMS320C30*, Wiley, New York, 1992.
3. R. Chassaing and D. W. Horning, *Digital Signal Processing with the TMS320C25*, Wiley, New York, 1990.
4. N. Kehtarnavaz and M. Keramat, *DSP System Design Using the TMS320C6000*, Prentice Hall, Upper Saddle River, NJ, 2001.
5. N. Kehtarnavaz and B. Simsek, *C6x-Based Digital Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 2000.
6. N. Dahnoun, *DSP Implementation Using the TMS320C6x Processors*, Prentice Hall, Upper Saddle River, NJ, 2000.
7. J. H. McClellan, R. W. Schafer, and M. A. Yoder, *DSP First: A Multimedia Approach*, Prentice Hall, Upper Saddle River, NJ, 1998.
8. C. Marven and G. Ewers, *A Simple Approach to Digital Signal Processing*, Wiley, New York, 1996.
9. J. Chen and H. V. Sorensen, *A Digital Signal Processing Laboratory Using the TMS320C30*, Prentice Hall, Upper Saddle River, NJ, 1997.
10. S. A. Tretter, *Communication System Design Using DSP Algorithms*, Plenum Press, New York, 1995.
11. A. Bateman and W. Yates, *Digital Signal Processing Design*, Computer Science Press, New York, 1991.
12. Y. Dote, *Servo Motor and Motion Control Using Digital Signal Processors*, Prentice Hall, Upper Saddle River, NJ, 1990.
13. J. Eyre, The newest breed trade off speed, energy consumption, and cost to vie for an ever bigger piece of the action, *IEEE Spectrum*, June 2001.
14. J. M. Rabaey, ed., VLSI design and implementation fuels the signal-processing revolution, *IEEE Signal Processing*, Jan. 1998.
15. P. Lapsley, J. Bier, A. Shoham, and E. Lee, *DSP Processor Fundamentals: Architectures and Features*, Berkeley Design Technology, Berkeley, CA, 1996.
16. R. M. Piedra and A. Fritsh, Digital signal processing comes of age, *IEEE Spectrum*, May 1996.
17. R. Chassaing, The need for a laboratory component in DSP education: a personal glimpse, *Digital Signal Processing*, Jan. 1993.
18. R. Chassaing, W. Anakwa, and A. Richardson, Real-time digital signal processing in education, *Proceedings of the 1993 International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Apr. 1993.
19. S. H. Leibson, DSP development software, *EDN Magazine*, Nov. 8, 1990.
20. D. W. Horning, An undergraduate digital signal processing laboratory, *Proceedings of the 1987 ASEE Annual Conference*, June 1987.
21. *TMS320C6000 Programmer's Guide*, SPRU198D, Texas Instruments, Dallas, TX, 2000.
22. *TMS320C6211 Fixed-Point Digital Signal Processor–TMS320C6711 Floating-Point Digital Signal Processor*, SPRS073C, Texas Instruments, Dallas, TX, 2000.
23. *TMS320C6000 CPU and Instruction Set Reference Guide*, SPRU189F, Texas Instruments, Dallas, TX, 2000.
24. *TMS320C6000 Assembly Language Tools User's Guide*, Texas Instruments, Dallas, TX,

SPRU186G, 2000.

25. *TMS320C6000 Peripherals Reference Guide*, SPRU190D, Texas Instruments, Dallas, TX, 2001.
26. *TMS320C6000 Optimizing Compiler User's Guide*, SPRU187G, Texas Instruments, Dallas, TX, 2000.
27. *TMS320C6000 Technical Brief*, SPRU197D, Texas Instruments, Dallas, TX, 1999.
28. *TMS320C64x Technical Overview*, SPRU395, Texas Instruments, Dallas, TX, 2000.
29. *TMS320C6x Peripheral Support Library Programmer's Reference*, SPRU273B, Texas Instruments, Dallas, TX, 1998.
30. *Code Composer Studio User's Guide*, SPRU328B, Texas Instruments, Dallas, TX, 2000.
31. *Code Composer Studio Getting Started Guide*, SPRU509, Texas Instruments, Dallas, TX, 2001.
32. *TMS320C6000 Code Composer Studio Tutorial*, SPRU301C, Texas Instruments, Dallas, TX, 2000.
33. *TLC320AD535C/I Data Manual Dual Channel Voice/Data Codec*, SLAS202A, Texas Instruments, Dallas, TX, 1999.
34. B. W. Kernigan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Upper Saddle River, NJ, 1988.
35. *Details on Signal Processing* (quarterly publication), Texas Instruments, Dallas, TX.
36. G. R. Gircys, *Understanding and Using COFF*, O'Reilly & Associates, Newton, MA, 1988.

第 2 章　DSK 的输入和输出

本章主要介绍利用 DSK 板上的 AD535 编解码器进行输入输出（附录 F 介绍了另一种利用 PCM3003 立体声编解码器进行输入输出的方法），并介绍一些 C 语言编程实例。

2.1　引言

DSP 技术典型应用的最基本系统如图 2.1 所示，它包括模拟输入和输出。在输入端有一个抗混叠滤波器，用来滤除信号中高于奈奎斯特频率（Nyquist frequency）的部分。奈奎斯特频率定义为抽样频率F_s的一半。如果信号带宽高于奈奎斯特频率就会产生频率混叠，这时频率高于$F_s/2$ 的信号和频率较低的信号将产生混叠现象。为了避免频率混叠现象，抽样定理要求抽样频率最小必须是信号中最高频率分量f的两倍，即：

$$F_s > 2f$$

也可以写成：

$$1/T_s > 2(1/T)$$

这里T_s是抽样周期。或者写成：

$$T_s < T/2$$

即抽样周期T_s 必须小于信号周期的一半。例如，人耳听不见 20 kHz 以上的频率信号，则可在输入端使用一个带宽或截止频率为 20 kHz 低通滤波器先进行滤波，去掉 20 kHz 以上的频率分量以避免频率混叠，然后使用大于 40 kHz 的抽样频率F_s来抽样音乐信号（抽样频率通常为 44.1 kHz 或 48 kHz）。图 2.2 表示了有混叠现象的信号，假设抽样频率F_s 是 4 kHz，即抽样周期T_s是 0.25 ms，这时就不能仅通过（0, 1, 0, −1）一系列抽样点来判断它是 1 kHz 信号还是 5 kHz 的信号。5 kHz 的信号看起来像 1 kHz 的信号，因此 1 kHz 信号是一个混叠信号，同样 9 kHz 信号也是 1 kHz 信号的混叠信号。

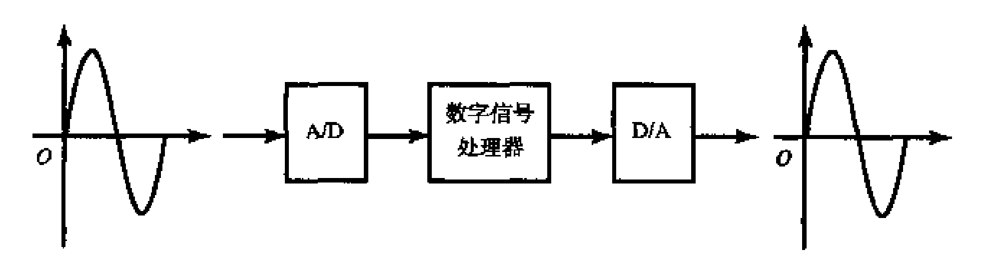


图 2.1　具有输入输出的 DSP 系统

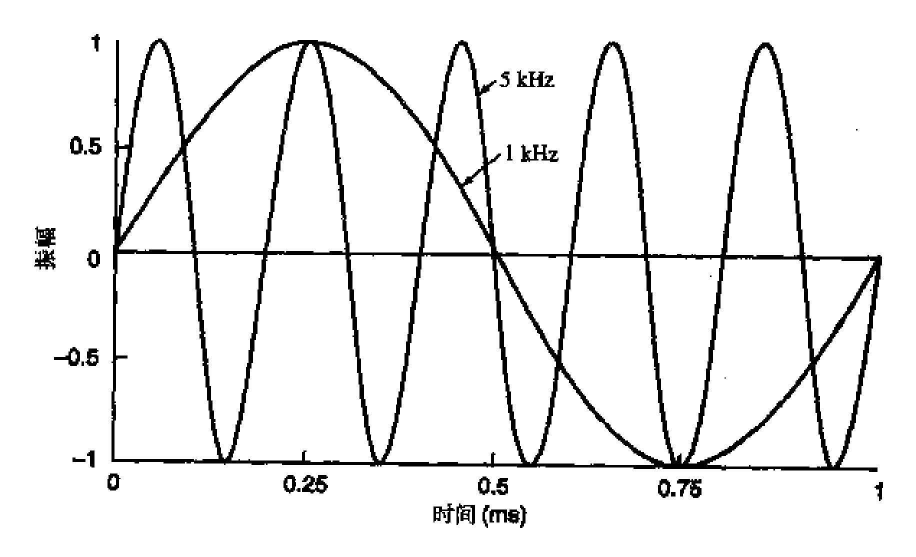


图 2.2　有频率混叠现象的正弦信号

2.2　利用 TLC320AD535 编解码器输入输出

在 DSK 板上有一个 TLC320AD535（AD535）编解码器，可用于输入输出信号。编解码器的 ADC 电路将输入的模拟信号转换成能被 DSP 处理的数字信号。输入信号的最大电平值由编解码器特定的 ADC 电路决定，DSK 板上编解码器输入的峰-峰值是 3 V。当输入信号处理完后，结果要输出到外部。如图 2.1 所示，在输出端有一个 DAC，它与 ADC 的作用正好相反，输出滤波器用来平滑和重建信号，ADC、DAC 和所需要的滤波功能都由 DSK 板上的编解码器 AD535 单个芯片来实现。

AD535 是一个基于 sigma-delta 技术的双通道话音/数据编解码器[1-5]，具有 ADC、DAC、低通滤波、过抽样等功能。AD535 编解码器有两个通道，抽样频率达 11.025 kHz。但 DSK 板上的编解码器只有一个输入和一个输出可供用户使用，用户可方便地利用两个 3.5 mm 音频电缆连接头连接输入输出端，抽样频率固定在 8 kHz 而不是 11.025 kHz[1]。

sigma-delta 转换器使用较低的抽样频率，但利用过抽样技术实现了较高的分辨率，这种 ADC 属于抽样频率比奈奎斯特频率高很多的一种 ADC。DSK 板上的 AD535 编解码器的过抽样因子是 64，用数字插值滤波器完成过抽样的功能。在这类器件中，量化噪声功率和抽样频率是相对独立的，其中调制器起到了噪声成形的作用，使噪声扩展到超出所关心的频率范围之外。噪声功率谱分布在 0 到 $F_s/2$ 之间，因此只有很小一部分噪声落在信号频带范围内。另外，还采用了数字滤波器去除带外噪声。

ADC 将输入信号转换成不连续的输出数据，这些数据用二进制补码表示，对应模拟信号相应时刻的抽样值。DAC 中有插值滤波器和数字调制器，抽取滤波器将数字信号的速率降低到与抽样速率一致，然后，DAC 的输出信号通过内部低通重建滤波器，最后得到模拟输出信号。通过使用 sigma-delta 调制器提供的噪声成形与过抽样技术，就可实现 ADC 和 DAC 低噪声的性能。

抽样频率 F_s 是通过编解码器上频率为 4096 kHz 的主时钟 MCLK 设定的，如下所示：

$$F_s = \text{MCLK}/512 = 8\ \text{kHz}$$

图 2.3 显示了 AD535 和 C6711 DSK 的接口框图，它包含在 CCS 软件包中。

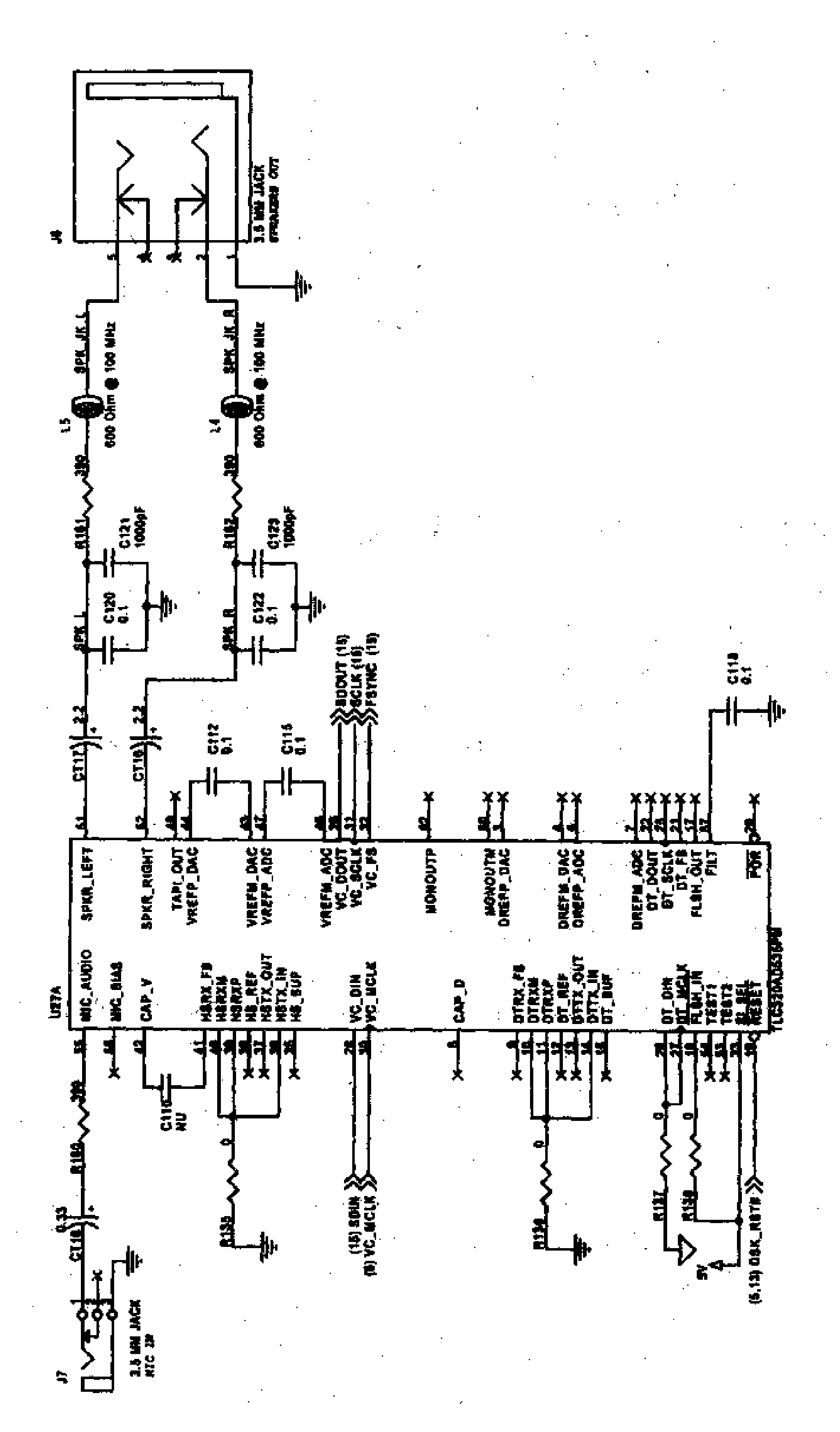


图 2.3 TLC320AD535 编解码器

接口之间利用串行方法进行通信，使用同一个串行口实现数据传输及传输控制，前者是主要的通信功能，后者是次要的通信功能，D/A 数据寄存器的最低有效位用于传输控制请求功能。

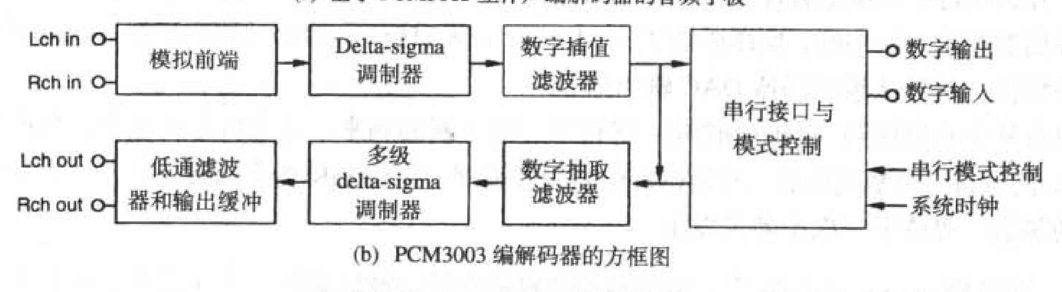
2.3 利用 PCM3003 立体声编解码器输入输出

附录 F[6]介绍了基于 PCM3003 立体声编解码器的音频子板，图 2.4(a)显示的是这种音频子板的照片，它的外形尺寸是 3×3.5 英寸①。

图 2.4(b)是 PCM3003 编解码器的方框图。这块子板的原理图在附录 F 中，可以将该子板插到 DSK 板上的 80 脚插头上。PCM3003 有两条完整的输入输出通道，其抽样率可编程控制，最大抽样频率是 72 kHz（TI 公司建议最大抽样率是 48 kHz）。附录 F 中有几个例子用来说明如何使用 PCM3003 的输入输出通道。



(a) 基于 PCM3003 立体声编解码器的音频子板



(b) PCM3003 编解码器的方框图

图 2.4 PCM3003 编解码器

2.4 C 程序编程实例

下面一些例子介绍使用 DSK 进行输入输出。通过学习这些例子，可更加熟悉使用硬件和软件，还可了解一些特定的应用背景。例如，在工程例子 sine2sliders 中，说明了如何使用两个滑动条；回声的例子介绍了改变缓存器长度对回声的影响；另一个回声例子则介绍了如何使用两个中断；方波产生的例子介绍了怎样产生方波并解释了 AD535 如何将数据转换为对应的输出电压。

例 2.1 使用中断的循环程序

本例介绍如何使用 AD535 进行输入输出，图 2.5 是 C 源程序 loop_intr.c，它是一个循环程序。与例 1.1 相同，它使用中断 INT11 作为中断源。

① 1 英寸=2.54 cm ——编者注。

```
//Loop_intr.c Loop program using interrupt, output is delayed input
//Comm routines and support files included in C6xdskinit.c

interrupt void c_int11()          //interrupt service routine
{
  int sample_data;

  sample_data = input_sample(); //input data
  output_sample(sample_data);   //output data
  return;
}

void main()
{
 comm_intr();                   //init DSK, codec, McBSP
 while(1);                      //infinite loop
}
```

图 2.5 使用中断的循环程序（loop_intr.c）

该程序很重要，因为可用它作为编写其他程序的基础。例如，为了实现数字滤波器，在输入和输出函数之间必须加入适当的算法。输入输出函数 input_sample 和 output_sample，以及 comm_intr 函数都包含在通信支持文件 c6xdskinit.c 中，这样做可使 C 源程序尽可能小。文件 c6xdskinit.c 可当做一个“黑匣子程序”，本书中的大部分例子都用到了它。

在完成初始化以及中断选择/启用使能后，执行 while 无限循环语句，直到中断发生。中断响应后，开始执行中断服务程序（ISR）c_int11，该程序是在矢量文件 vectors_11.asm 中指定的。每个抽样周期产生一次中断，抽样周期 $T_s = 1/F_s = 1/(8\ \text{kHz}) = 0.125$ ms。这时，从编解码器 ADC 读入抽样值，然后从编解码器 DAC 输出信号。

执行从中断返回到 while(1)语句，等待下一次中断的到来。注意可用处理程序替换无穷循环 while(1)的等待。中断响应后，开始执行中断服务程序，执行 ISR 指定的必要任务，然后返回正在调用的函数，等待下一次中断的发生。

1. 在函数 output_sample 中，输出数据的最低有效位被屏蔽掉，用于次要的通信和传输。编解码器 AD535 的 DAC 是一个有 15 位有效位的器件，因此它使用 16 位字的最高 15 个有效位作为输出数据，最低有效位（LSB）用于控制功能。在函数 output_sample 中，16 位输出数据的最低有效位被屏蔽掉，通知编解码器不必等待后面的控制数据。
2. 在函数 comm_intr 中，执行了下面的任务：
   (a) 初始化 DSK。
   (b) 配置/选择 INT11 和传输中断 XINT0。
   (c) 使能指定中断。
   (d) 使能全局中断位（GIE）。
   (e) 访问多通道缓冲串行口（McBSP）0。

被调用执行上述任务的中断函数包含在 CCS 的 c6xinterrupts.h 文件中。

创建并建立名为 loop_intr 的工程，使用与例 1.1 相同的支持文件。辅助材料中包含了本书所

有的源文件和支持文件，其他需要的支持文件也包含在CCS中。从IN连接头J7输入一个正弦波，其振幅大约是1~2 V（峰-峰值），频率大约是1~3 kHz，连接DSK输出OUT连接头J6，可以观察到与输入相同频率的单音，只是振幅有些减小。使用示波器观察，可以发现输出波形相对输入波形有些延迟。将输入正弦波的幅度超出3 V（峰-峰值）以上，观察到波形开始失真了。

例2.2　使用轮询的循环程序

本例使用轮询方式实现每个抽样周期T_s输入和输出样点的循环程序，而例2.1程序loop_intr.c是中断驱动程序，这里C程序loop_poll.c（如图2.6所示）实现了该轮询循环程序。轮询技术连续不断地测试什么时候数据已准备好。和中断技术相比，它虽然比较简单，但效率不是很高。

```
//loop_poll.c Loop program using polling, output is delayed input
//Comm routines and support files included in C6xdskinit.c

void main()
{
  int sample_data;

  comm_poll();                               //init DSK, codec, McBSP
  while(1)                                   //infinite loop
  {
    sample_data = input_sample();  //input sample
    output_sample(sample_data);    //output sample
  }
}
```

图2.6　使用轮询的循环程序（loop_poll.c）

1. 在函数input_sample中，调用另一个函数mcbsp0_read，它读取输入到ADC的数据，这些数据来自多通道缓冲串行口（McBSP）0，或简称为SP0的数据接收寄存器（DRR）。串行口控制寄存器（SPCR）首先与0x2进行AND操作，测试SPCR的接收准备寄存器（RRDY）位1是否置位，如图B.8所示。
2. 在函数output_sample中，调用另一个函数mcbsp0_write，把从DAC的输出写入到McBSP 0（SP0）的数据传送寄存器（DXR）。SPCR首先和0x20000进行AND操作，测试SPCR发送准备寄存器（XRDY）位17是否置位。程序在无限循环while(1)中等待数据准备好，再开始执行输出。

除了矢量文件vectors_11.asm外，其他支持文件和例2.1或例1.1都相同，可以把vectors_11.asm（使用INT11）替换为vectors.asm（在辅助材料中），或按如下方法编辑文件vectors_11.asm：

1. 删除.ref_c_int11语句，这是汇编器命令，它交叉引用中断服务程序（ISR）_c_int11。在函数前面的第一个下划线是C函数约定的规则。
2. 用NOP（空指令）替换指令b_c_int11，这条指令原来指向ISR。

创建并建立工程loop_poll，使用例2.1同样的输入，检验输出结果是否一致。

例2.3　使用轮询方式产生正弦信号

本例利用4个数据点产生正弦波形，进一步介绍轮询方法的使用。图2.7给出了利用4个点产生正弦波的C源程序sine4_poll.c。

```
//Sine4_poll.c Sine generation using 4 points; f=Fs/(# points)=2 kHz

int loop = 0;
short sine_table[4] = {0,1000,0,-1000};          //sine values
short amplitude = 1;                              //for slider

void main()
{
  int sample_data;

  comm_poll();                                    //init DSK, codec, McBSP
  while(1)                                        //infinite loop
  {
   sample_data = (sine_table[loop]*amplitude);    //scaled value
   output_sample(sample_data);                    //output sine value
   if (loop < 3) ++loop;                          //increment index
   else loop = 0;                                 //reinit @ end of buffer
  }
}
```

图 2.7 用轮询方法和 4 个数据点产生正弦波的程序（sine4_poll.c）

使用和例 2.2（同样可参见例 1.1）中 loop_poll 同样的支持文件，在每个抽样周期 $T_s = 1/F_s$，输出由缓冲区（表）sine_table 的数据组成，数据为 0, 1000, 0, –1000, 0, 1000，…，每隔 0.125 ms 输出一个。

建立并运行工程 sine4_poll，检验输出是否是偏置为 1 V，频率为 $f = F_s$/点数 = 8 kHz/4 = 2 kHz 的正弦波形，产生偏置是由于 AD535 编解码器自身特点。

装载 GEL 文件 sine4_poll.gel（如图 2.8 所示），和例 1.1 一样，访问滑动条函数 amplitude。把滑动条从位置 1 变到位置 2, 3, …, 10，并检验输出波形信号幅度是否增加了。

```
/*Sine4_poll.gel Create slider and vary amplitude of sine wave*/

menuitem "Sine Amplitude"

slider Amplitude(1,10,1,1,amplitudeparameter) /*incr by 1,up to 10*/
{
    amplitude = amplitudeparameter;       /*vary amplitude of sine*/
}
```

图 2.8 说明滑动条函数的 GEL 文件（sine4_poll.gel）

将滑动条函数 amplitude 变化范围改成 30 到 90（代替 10），每次步进值仍为 1，编辑 GEL 文件，把它存到 sine4_poll.gel 文件中。重载该程序，并通过 GEL 访问它，当滑动条的位置在 32 时，输出电压幅度大约为 2.7 V（峰-峰值），这时正弦值为+32 000 和–32 000。将滑动条的位置调高到 33, 34, …, 65。观察到当滑动条在位置 65 时，输出信号幅度降到大约为 0.1 V（峰-峰值）。当滑动条在位置 66, 67, …, 90 时，波形的幅度会不会重新增加？

例 2.4 用两个滑动条控制幅度和频率，产生正弦信号

程序 sine2sliders.c（如图 2.9 所示）采用轮询方法控制输出速率，产生正弦波信号。它利用两

个滑动条分别改变正弦波的幅度和频率，并利用 32 点的查找表，通过每个周期选择不同数目的数据点数来实现频率的改变。幅度滑动条用来改变输出信号的幅度，图 2.10 给出了合适的 GEL 文件 sine2sliders.gel。

```
//Sine2sliders.c Sine generation with different # of points

short loop = 0;
short sine_table[32]={0,195,383,556,707,831,924,981,1000,
                      981,924,831,707,556,383,195,
                      0,-195,-383,-556,-707,-831,-924,-981,-1000,
                     -981,-924,-831,-707,-556,-383,-195}; // sine data
short amplitude = 1;                            //for slider
short frequency = 2;                            //for slider

void main()
{
 comm_poll();                                   //init DSK, codec, McBSP
 while(1)                                       //infinite loop
  {
   output_sample(sine_table[loop]*amplitude);   //output scaled value
   loop += frequency;                           //incr frequency index
   loop = loop % 32;                            //modulo 32 to reset
  }
}
```

图 2.9 用两个滑动条控制幅度和频率，并产生正弦波的程序（sine2sliders.c）

```
/*Sine2sliders.gel Two sliders to vary amplitude and frequency*/

menuitem "Sine Parameters"

slider Amplitude(1,8,1,1,amplitudeparameter) /*incr by 1,up to 8*/
       {
        amplitude = amplitudeparameter;      /*vary amplitude*/
       }

slider Frequency(2,8,2,2,frequencyparameter) /*incr by 2,up to 8*/
       {
        frequency = frequencyparameter;      /*vary frequency*/
       }
```

图 2.10 控制正弦信号幅度和频率的两个滑动条函数 GEL 文件（sine2sliders.gel）

查找表或缓存区中的32个数值分别对应于sin(t)，且t分别为0, 11.25, 22.5, 33.75, 45,…, 348.75度的值（这些值再和 1000 相乘）。频率滑动条位置从 2 变到 8，步长是 2。模操作用于测试当指针指向正弦波缓冲区的末尾时输出变化的情况。当循环变量达到 32 时，就被重新初始化为 0。例如，当频率滑动条在位置 2 时，循环或频率变量在查找表中每隔一个数据步进一次，这样每个周期对应 16 个数据点。

建立工程 sine2sliders，使用和例 2.3 同样的支持文件，检验产生的信号频率是否为 $f = F_s/16 =$ 500 Hz。将滑动条的位置分别提高到 4, 6 和 8，检验产生的信号频率是否分别为 1000 Hz, 1500 Hz

和 2000 Hz。注意当滑动条的位置为 4 时，循环或频率变量从查找表中选择 8 个数据（每个周期）：sin[0], sin[4], sin[8], …, sin[28]，对应的数据分别为 0, 707, 1000, 707, 0, –707, –1000 和–707，产生信号的频率为 $f=F_s/8=1\,\text{kHz}$（和例 1.1 一样）。

例 2.5　将输入数据存储在缓冲区的循环程序

程序 loop_store.c（如图 2.11 所示）是基于中断的程序，中断源 INT11 每产生一次中断，就从编解码器的 ADC 和 DAC 读入和输出一个抽样值，另外，每个抽样值被写到 512 个元素的循环缓冲区中，该循环缓冲区用数组缓冲区来实现，其索引变量随存储抽样值的增加而增加。当索引变量增加到 512 时，它重新复位为 0，因此，该数组总是只含有 512 个最近抽样值。

```
//Loop_store.c Data acquisition. Input data also stored in buffer

#define BUFFER_SIZE 512              //buffer size
short buffer[BUFFER_SIZE];           //buffer buffer
short i = 0;

interrupt void c_int11()             //interrupt service routine
{
 int sample_data;

 sample_data = input_sample();       //new input data
 output_sample(sample_data);         //output data

 buffer[i] = sample_data;            //store data in buffer
 i++;                                //increment buffer index
 if (i == BUFFER_SIZE) i = 0;        //reinit index if buffer full
 return;                             //return from ISR
}

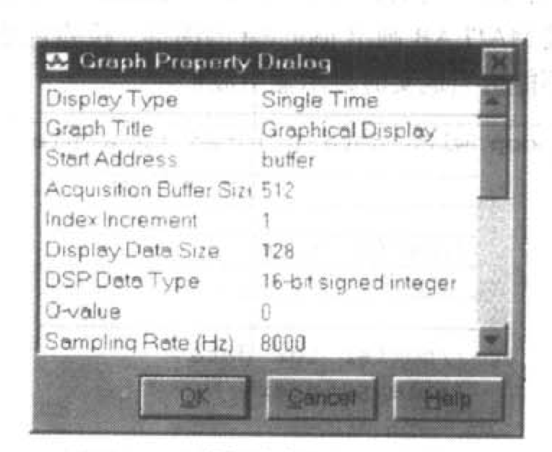
void main()
{
 comm_intr();                        //init DSK, codec, McBSP
 while(1);                           //infinite loop
}
```

图 2.11　在存储器中输入/输出数据的循环程序（loop_store.c）

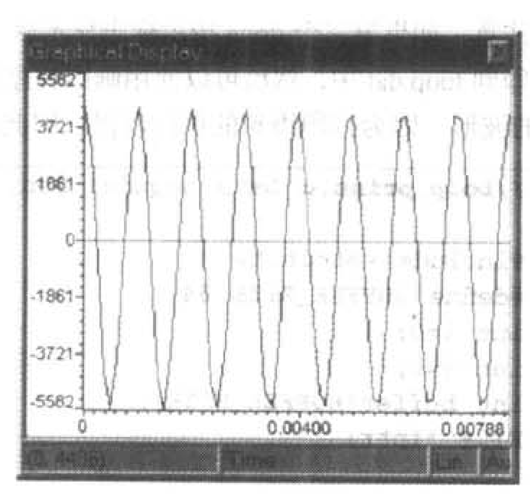
建立名为 loop_store 的工程，输入幅度大约为 0.5 V（峰-峰值）、频率为 1 kHz 的正弦信号，运行并检验输出结果。

用 CCS 画出输入数据波形，包括时域和频域图（参见例 1.2）。选择菜单 View→Graph→Time/Frequency，用 buffer 作为起始地址。为了获得清晰的图形，选择显示数据的个数为 128 个点（而不是 512 点），如图 2.12(a)的 Graph Property Dialog 窗口所示（其他的选项为默认值），检查用 CCS 画出的 1 kHz 正弦时域波形，如图 2.12(b)所示。

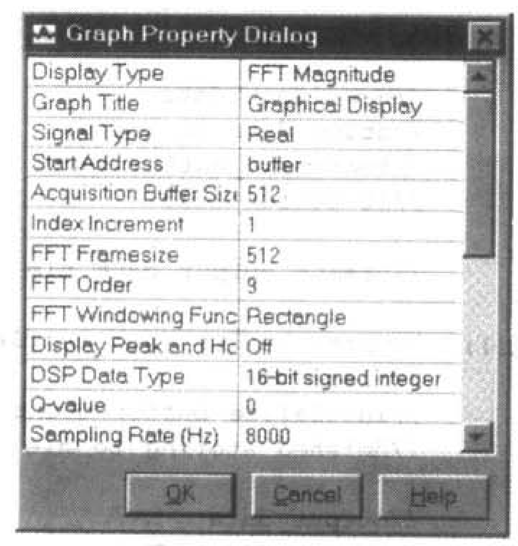
用鼠标右击图形窗口，或再选择菜单 View→Graph→Time/Frequency，在 Graph Property Dialog 窗口选择显示类型 FFT magnitude，如图 2.12(c)所示，得到输入数据频域图形。注意 FFT 的阶数 $M=9$，因为 $2^M=512$。在图 2.12(d)中，1 kHz 处的脉冲表示 1 kHz 的正弦波形。



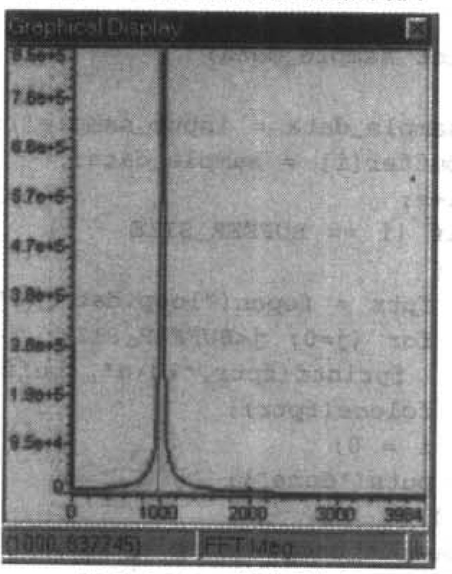
(a) 时域图的参数设置图



(b) 表示 1 kHz 信号的存储输出数据时域图



(c) FFT 幅度图的参数设置图



(d) 表示 1 kHz 正弦信号的存储输出数据 FFT 幅度图

图 2.12 程序 loop_store 的 CCS 图

例 2.6 将缓冲区的数据打印到文件的循环程序

本例扩展了前面的循环程序，将存储缓冲区中的输入输出数据打印到文件中，图 2.13 给出了实现该工程例子的 C 源程序 loop_print.c。在程序中，执行 printf 语句需要很长时间（4000 个时钟周期量级），而使用实时数据传送（RTDX）可减少到 30 个时钟周期，附录 G 对此进行了介绍。

完成 DSK 的初始化后，puts 语句打印出单词 start 作为指示，然后进入无限 while 循环中。每次中断响应后，进入 ISR，新采集到的一个数据就被保存到大小为 64 的数据缓冲区中。

每当存储一个新抽样数据时，缓冲区索引变量就会增加 1。当到达缓冲区的末尾时，指示缓冲区满，这时打开 loop.dat 文件，缓冲区的内容被写入到该文件中，然后在 CCS 命令窗口显示 done 作为指示。不断重复该过程，然后采集一组新的 64 个数据，done 指示标志会再一次显示（当每组数据存满缓冲区并且存入文件 loop.dat 后）。

建立并运行工程 loop_print，输入频率为 1 kHz 且幅度为 1 V（峰-峰值）的正弦信号，当指示标志 done 显示时，停止程序执行。缓冲区中表示正弦波的 64 个输入数据可从文件 loop.dat 恢复

出来。注意：如果第三个 done 指示标志显示后，停止程序执行，第三组 64 个数据就存储到缓冲区并打印到 loop.dat 中，然后可以使用画图程序或 MATLAB 画出 loop.dat 的图形，检验它是否是 1 kHz 的波形。因为打印语句的执行较慢，因此不能恰当地实时显示输出波形。

```
//Loop_print.c Data acquisition. Loop with data printed to a file

#include <stdio.h>
#define BUFFER_SIZE 64                      //buffer size
int i=0;
int j=0;
int buffer[BUFFER_SIZE];                    //buffer for data
FILE *fptr;                                 //file pointer

interrupt void c_int11()                    //interrupt service routine
{
 int sample_data;

 sample_data = input_sample();              //new input data
 buffer[i] = sample_data;                   //store data in buffer
 i++;                                       //increment buffer count
 if (i == BUFFER_SIZE - 1)                  //if buffer full
  {
   fptr = fopen("loop.dat","w");            //create output data file
   for (j=0; j<BUFFER_SIZE; j++)
     fprintf(fptr,"%d\n", buffer[j]);       //write buffer data to file
   fclose(fptr);                            //close file
   i = 0;                                   //initialize buffer count
   puts("done");                            //finished storing to file
  }
 output_sample(sample_data);                //output data
 return;                                    //return from ISR
}

void main()
{
 comm_intr();                               //init DSK, codec, McBSP
 puts("start\n");                           //print "start" indicator
 while(1);                                  //infinite loop
}
```

图 2.13 输入/输出数据存储到缓冲区，再输出到文件中的循环程序（loop_print.c）

例 2.7 使用查表法产生方波

本例使用查表法产生一个方波，并且介绍了 AD535 的数据格式。图 2.14 给出了实现该工程的程序 squarewave.c。建立大小为 256 的整型数据缓冲区，在 main 函数中，在缓冲区查找表中存入数据：前半部分存入$(2^{15}-1)=32\ 767$，后半部分存入$-2^{15}=-32\ 768$。当每个抽样周期T_s响应中断时，将缓冲区中的一个数据送到输出端；当表中的一个数据送到输出端后，程序执行返回到无限 while 循环，等待下一次中断产生并输出表中的下一个数据；当到达缓冲区（表）的末尾时，缓冲区的索引变量会被重新初始化并指向缓冲区的开始。

```
//Squarewave.c Generates a squarewave using a look-up table

#define table_size (int)0x100                //size of table = 256
int data_table[table_size];                  //data table array
int i;

interrupt void c_int11()                     //interrupt service routine
{
 output_sample(data_table[i]);               //output value each Ts
 if (i < table_size) ++i;                    //if table size is reached
 else i = 0;                                 //reinitialize counter
 return;                                     //return from interrupt
}

main()
{
 for(i=0; i<table_size/2; i++)               //set 1st half of buffer
      data_table[i] = 0x7FFF;                //with max value (2^15)-1
 for(i=table_size/2; i<table_size; i++)      //set 2nd half of buffer
      data_table[i] = -0x8000;               //with -(2^15)

 i = 0;                                      //reinit counter
 comm_intr();                                //init DSK, codec, McBSP
 while (1);                                  //infinite loop
}
```

图 2.14 产生方波的程序（squarewave.c）

建立并运行工程 squarewave，检验输出是峰-峰值为 2.8 V、偏置约 1.1 V 的方波信号。

注意，因为编解码器是 16 位的，其有效输入数据在-2^{15}和$2^{15}-1$之间，即-32 768 和 32 767。将表中前半部分的值用 0x8000 = 32 768 替换 0x7FFF = 32 767，重新建立并运行工程，检验一下，这时将不会再产生方波信号。

例 2.8 使用查表法产生锯齿波

图 2.15 给出了使用查表法产生锯齿波的程序 ramptable.c，建立大小为 1024 的整型数据缓冲区，在 main 函数中，将后面 1024 个数据存入缓冲区查找表中：0, 0x20, 0x40, …，也就是 0, 32, 64, …, 32 736（十进制）。

建立并运行工程 ramptable，检验输出将产生一个锯齿波，该锯齿波的峰值相对偏移 1.1 V，随着输入 32, 64, …等数据而减小，这是因为 AD535 编解码器的数据格式是二进制补码，因此产生的锯齿波为负斜率，峰-峰值大约是 1.4 V。

用-0x20 代替 0x20，检验这时产生的锯齿波是正斜率的，峰-峰值为 1.4 V。锯齿波从偏置值大约 1.1 V 开始，增加到约 2.5 V。

例 2.9 不用查表法产生锯齿波

例 2.8 通过在查找表中存入一组数据，然后在每个抽样周期输出查找表中一个数据，当到达查找表的末尾时再返回查找表开始的数据。图 2.16 给出了与例 2.8 使用不同方法产生锯齿波的程序 ramp.c，开始时初始输出值为 0，输出数据值按每个抽样周期T_s增加 0x20，这样输出的数据值就是 0, 32, 64, 96, …, 32 736。

```
//Ramptable.c Generates a ramp using a look-up table

#define table_size (int)0x400    //size of table=1024
int data_table[table_size];      //data table array
int i;

interrupt void c_int11()         //interrupt service routine
{
 output_sample(data_table[i]);   //ramp value for each Ts
 if (i < table_size-1) i++;      //if table size is reached
 else i = 0;                     //reinitialize counter
 return;                         //return from interrupt
}

main()
{
  for(i=0; i < table_size; i++)
   {
    data_table[i] = 0x0;         //clear each buffer location
    data_table[i] = i * 0x20;    //set to 0,32,64,96, ... ,32736
   }
  i = 0;                         //reinit counter
  comm_intr();                   //init DSK, codec, McBSP
  while (1);                     //infinite loop
}
```

图 2.15 使用查找表产生锯齿波的程序（ramptable.c）

```
//Ramp.c Generates a ramp

int output;

interrupt void c_int11() //interrupt service routine
{
 output_sample(output);  //output for each sample period
 output += 0x20;         //incr output value
 if (output == 0x8000)   //if peak is reached
 output = 0;             //reinitialize
 return;                 //return from interrupt
}

void main()
{
 output = 0;             //init output to zero
 comm_intr();            //init DSK, codec, McBSP
 while(1);               //infinite loop
}
```

图 2.16 锯齿波产生程序（ramp.c）

建立并运行工程 ramp，检验输出结果是否和例 2.8 一样，产生一个负斜率的锯齿波。为了得到一个正斜率的锯齿波，将输出改为：

```
output -= 0x20
```

这样输出变为 0, −32, −64, …, −32 736。同样改变重新初始化 output 的 if 语句，即：

```
if(output == -0x8000)
```

检验输出是否是一个正斜率的锯齿波。

例 2.10 回声

图 2.17 给出了产生输入信号回声的程序 echo.c，缓冲区长度或大小决定回声的效果，2000 个数据点的缓冲区只能产生清晰的回声，而 16 000 点数据的缓冲区将产生太长的延迟，它产生的效果更像是重复。输出由刚刚得到的抽样值和最早存储在缓冲区中的抽样值的和组成。如果缓冲区太小，最新抽样值和最早抽样值的时间延迟太短就听不到回声产生的效果，将最早抽样值做一定的衰减可增强回声的效果。

```
//Echo.c Echo effect changed with size of buffer (delay)
short input, output;
short bufferlength = 3000;                        //buffer size for delay
short buffer[3000];                               //create buffer
short i = 0;
short amplitude = 5;                              //to vary amplitude of echo

interrupt void c_int11()                          //ISR
{
 input = input_sample();                          //newest input sample data
 output=input + 0.1*amplitude*buffer[i];          //newest sample+oldest sample
 output_sample(output);                           //output sample

 buffer[i] = input;                               //store newest input sample
 i++;                                             //increment buffer count
 if (i >= bufferlength) i = 0;                    //if end of buffer reinit
}

main()
{
 comm_intr();                                     //init DSK, codec, McBSP
 while(1);                                        //infinite loop
}
```

图 2.17 产生回声（echo.c）

当采集到新的抽样值时，将它存储在 x 存储单元，这时输出就变成新抽样值和最早存储在存储单元 $x+1$ 的抽样值的和，这里 $x = 0, 1, 2, \cdots, 2998$。当缓冲区索引到缓冲区的末尾时（buffer[2999]），新得到的抽样存储在这里，最早的抽样值在缓冲区的开始。

建立并运行工程 echo，波形文件 theforce.wav（包含在辅助材料中）可以用做输入，可用共享软件 Goldwave（在附录 E 中有介绍）连续不断地循环播放该文件。

把缓冲区的大小从 1000 个数据改为 8000 个数据，可以观察到较大的缓冲区会使最新的数据和最早的数据产生较大的延迟。一个 GEL 文件（在辅助材料中）可用来增加或减小幅度或回声的效果。

如果将输出（代替输入）存储在缓冲区中，回声效果会逐渐消失。使用：

buffer[i] = output

重新建立并运行程序，检验回声效果是否逐渐消失。

例 2.11 使用两个中断，产生不同效果的回声

本例扩展了例 2.10，增加了一些附加的回声效果。它采用另一种方法，用两个中断进行读写。使用三个滑动条控制最早输入抽样值的幅度，改变缓冲区的大小产生不同的延迟效果以及回声逐渐消失的效果。图 2.18 给出了实现该工程的程序 echo_control.c。和前面的例子一样，它使用发送中断 INT11 写入数据，另外，它还使用接收中断 INT12 读入数据。除了下面一些改动外，支持文件也和前面的例子相同。

1. 修改文件 c6xdskinit.c，以控制两个中断。在文件 c6xdskinit.c 中添加下面两行程序代码：

```
config_Interrupt_Selector(12,RINT0); //receive INT12
enableSpecificInt(12);               //interrupt 12
```

2. 修改矢量文件 vectors_11.asm，添加和 INT12 有关的指向中断服务程序 c_int12 的分支指令，文件 vectors_11_12.asm 中也进行这种修改。

```
//Echo_control.c Echo using two interrupts for read and write
//3 sliders to control effects: buffer size, amplitude, fading

short input, output;
short bufferlength = 1000;                 //initial buffer size
short i = 0;                               //buffer index
short buffer[8000];                        //max size of buffer
short delay = 1;                           //determines size of buffer
short delay_flag = 1;                      //flag if buffer size changes
short amplitude = 1;                       //amplitude control by slider
short echo_type = 0;                       //no fading (1 for fading)

interrupt void c_int11()                   //ISR INT11 to write
{
 short new_count;                          //count for new buffer

 output=input+0.1*amplitude*buffer[i];     //newest+oldest sample
 if (echo_type == 1)                       //if fading is desired
  {
   new_count = (i-1) % bufferlength;       //previous buffer location
   buffer[new_count] = output;             //to store most recent output
  }
 output_sample(output);                    //output delayed sample
}

interrupt void c_int12()                   //ISR INT12 to read
{
 input = input_sample();                   //newest input sample data
 if (delay_flag != delay)                  //if delay has changed
  {                                        //->new buffer size
```

```
    delay_flag = delay;                   //reint for future change
    bufferlength = 1000*delay;            //new buffer length
    i = 0;                                //reinit buffer count
   }
  buffer[i] = input;                      //store input sample
  i++;                                    //increment buffer index
  if (i == bufferlength) i=0;             //if @ end of buffer reinit
 }

main()
{
  comm_intr();                            //init DSK, codec, McBSP
  while(1);                               //infinite loop
}
```

图 2.18　具有不同控制效果的回声产生程序（echo_control.c）

和例 2.10 一样，使用相同的文件 theforce.wav 作为输入（在辅助材料中）。在每个抽样周期产生一次中断：首先 INT11 写入数据，然后 INT12 读出数据，输出是最新输入抽样和最早抽样的和。

1. 建立并运行工程 echo_control。
2. 访问三个滑动条：振幅、延迟和类型，图 2.19 给出了 GEL 文件 echo_control.gel。将振幅设置滑动条设置在位置 5，延迟滑动条设置在位置 3，因为延迟不等于 delay_flag，所以缓冲区的大小已经改变了，新的缓冲区大小设置为 $1000\times3=3000$，这两个滑动条的设置对应于例 2.10 中的相同条件。延迟滑动条可取值 1, 2, …, 8，这样允许将缓冲区的大小设置为 1000, 2000, 3000, …, 8000。提高延迟滑动条的值到 4 和 5，延长最新数据和最早数据之间的延迟，观察回声效果。
3. 类型滑动条在位置 1 时，将产生回声逐渐消失的效果，因为输出变成最近的输出。为了获得清晰的回声逐渐消失的效果，暂时停止播放输入的 theforce.wav 文件。

用三个滑动条进行实验，会产生不同的回声效果。

```
//Echo_control.gel Sliders vary time delay, amplitude, and type of echo

menuitem "Echo Control"

slider Amplitude(1,8,1,1,amplitude_parameter)   /*incr by 1, up to 8*/
 {
  amplitude = amplitude_parameter;               /*vary amplit of echo*/
 }
slider Delay(1,8,1,1,delay_parameter)           /*incr by 1, up to 8*/
 {
  delay = delay_parameter;                       /*vary delay of echo*/
 }
slider Type(0,1,1,1,echo_typeparameter)         /*incr by 1, up to 1*/
 {
  echo_type = echo_typeparameter;                /*echo type for fading*/
 }
```

图 2.19　控制回声振幅、延迟和衰变的 GEL 文件（echo_control.gel）

例 2.12　用程序中产生的数据表产生正弦波

本例产生正弦信号一个周期的数据，这些数据用来建立一张查找表，然后输出这些数据，产生正弦波形。图 2.20 给出了实现该工程的程序 sinegen_table.c，它产生频率 $f = F_s$/点数 $= 8000/10 = 800$ Hz 的正弦信号。

建立并运行 sinegen_table 工程，在该工程中使用了发送中断 INT11，检验产生的正弦波频率是否为 800 Hz。改变数据点数，产生一个 400 Hz 的正弦波（只需改变 table_size）。

```
//Sinegen_table.c Generates a sinusoid for a look-up table

#include <math.h>
#define table_size (short)10      //set table size
short sine_table[table_size];     //sine table array
short i;

interrupt void c_int11()          //interrupt service routine
{
 output_sample(sine_table[i]);    //output each sine value
 if (i < table_size - 1) ++i;     //incr index until end of table
   else i = 0;                    //reinit index if end of table
 return;                          //return from interrupt
}

void main()
{
float pi=3.14159;

for(i = 0; i < table_size; i++)
  sine_table[i]=10000*sin(2.0*pi*i/table_size); //scaled values

i = 0;
comm_intr();                      //init DSK, codec, McBSP
while(1);                         //infinite loop
}
```

图 2.20　在程序产生查找表，进而产生正弦波的程序（sinegen_table.c）

例 2.13　使用 MATLAB 建立的查找表产生正弦波

本例介绍了使用 MATLAB 产生查表法，进而产生正弦波的过程。图 2.21 给出了 MATLAB 程序 sin1500.m，它产生了包含 24 个周期 128 个数据点的文件，生成的正弦波频率是 $f = F_s \times$ 周期数/总点数 $= 1500$ Hz。

在 MATLAB 中运行 sin1500.m，检验包含 128 个数据点的头文件 sin1500.h 是否和图 2.22 所示一致。用不同的点数表示不同的正弦信号频率，这一点只需在 MATLAB 程序 sin1500.m 中做很小改动就可很方便地实现。

图 2.23 给出了实时实现该工程的 C 程序 sin1500MATL.c，它包含 MATLAB 创建的头文件。也可参见例 2.12，它在 C 源程序 main 中产生数据表，而不是在 MATLAB 中产生数据表。

建立并运行工程 sin1500MATL，检验输出是否是 1500 Hz 的正弦波信号。注意当在 CCS 中查看头文件 sin1500.h 时，要仔细一些，避免把头文件截断。

```
%sin1500.m Generates 128 points representing sin(1500) Hz
%Creates file sin1500.h
for i=1:128
  sine(i) = round(1000*sin(2*pi*(i-1)*1500/8000)); %sin(1500)
end

fid = fopen('sin1500.h','w');                %open/create file
fprintf(fid,'short sin1500[128]={');         %print array name,"={"
fprintf(fid,'%d, ' ,sine(1:127));            %print 127 points
fprintf(fid,'%d' ,sine(128));                %print 128th point
fprintf(fid,'};\n');                         %print closing bracket
fclose(fid);                                 %close file
```

图 2.21　用 MATLAB 产生正弦波查找表的程序（sin1500.m）

```
short sin1500[128]={0, 924, 707, -383, -1000, -383, 707, 924, 0,
-924, -707, 383, 1000, 383, -707, -924, 0, 924, 707, -383,
-1000, -383, 707, 924, 0, -924, -707, 383, 1000, 383, -707,
-924, 0, 924, 707, -383, -1000, -383, 707, 924, 0, -924, -707,
383, 1000, 383, -707, -924, 0, 924, 707, -383, -1000, -383, 707,
924, 0, -924, -707, 383, 1000, 383, -707, -924, 0, 924, 707,
-383, -1000, -383, 707, 924, 0, -924, -707, 383, 1000, 383,
-707, -924, 0, 924, 707, -383, -1000, -383, 707, 924, 0, -924,
-707, 383, 1000, 383, -707, -924, 0, 924, 707, -383, -1000,
-383, 707, 924, 0, -924, -707, 383, 1000, 383, -707, -924, 0,
924, 707, -383, -1000, -383, 707, 924, 0, -924, -707, 383, 1000,
383, -707, -924};
```

图 2.22　MATLAB 产生的正弦波查找表头文件（sin1500.h）

```
//Sin1500MATL.c Generates sine from table created with MATLAB

#include "sin1500.h"            //sin(1500) created with MATLAB
short i=0;

interrupt void c_int11()
{
 output_sample(sin1500[i]);     //output each sine value
 if (i < 127) ++i;              //incr index until end of table
   else i = 0;
 return;                        //return from interrupt
}

void main()
{
 comm_intr();                   //init DSK, codec, McBSP
 while(1);                      //infinite loop
}
```

图 2.23　产生正弦波的程序，该程序利用 MATLAB 产生的数据作为头文件（sin1500MATL.c）

例 2.14 幅度调制

本例介绍了幅度调制（AM）的实现方案，图 2.24 给出了产生 AM 信号的程序 AM.c。缓冲区 baseband 含有 20 个数据点，表示频率 $f = F_s/20 = 400$ Hz 的基带余弦信号；缓冲区 carrier 也有 20 个数据点，表示频率 $f = F_s \times$ 周期数/总点数 $= F_s$/每周期的点数 $= 2$ kHz 的载波信号；变量 amp 用来改变调制信号的幅度。AM.c 源程序不是中断驱动的，需要选择合适的矢量支持文件。

建立和实现工程 AM，检验输出信号中包含 2 kHz 的载波信号和两个边带信号。边带信号的频率等于载波信号加或减边带信号的频率，也就是 1600 Hz 和 2400 Hz 信号。

```
//AM.c AM using table for carrier and baseband signals

short amp = 1;

void main()
{
 short baseband[20]={1000,951,809,587,309,0,-309,-587,-809,-951,
       -1000,-951,-809,-587,-309,0,309,587,809,951}; //400-Hz baseband
 short carrier[20] ={1000,0,-1000,0,1000,0,-1000,0,1000,0,
       -1000,0,1000,0,-1000,0,1000,0,-1000,0}; //2-kHz carrier
 short output[20];
 short k;

 comm_poll();                              //init DSK, codec, McBSP
 while(1)                                  //infinite loop
  {
   for (k=0; k<20; k++)
    {
     output[k]= carrier[k] + ((amp*baseband[k]*carrier[k]/10)>>12);
     output_sample(20*output[k]);          //scale output
    }
  }
}
```

图 2.24 幅度调制程序（AM.c）

加载 GEL 文件 AM.gel，增大变量 amp 的值，检验被调制的基带信号。注意载波和基带信号的积（在输出等式内）被除以 2^{12}（右移 12 位）。话音扰乱器（见例 4.9）进一步利用调制以达到扰乱输入信号的目的。

使用外部输入作为边带信号的调幅方法

程序 AM_extin.c（在辅助材料中）介绍了另外一种获得 AM 信号的方法，它利用外部输入信号作为边带信号，用查表方法产生 2 kHz 的载波信号。

建立工程 AM_extin，使用幅度小于 0.35 V、频率低于 2 kHz 的信号作为正弦边带信号，检验该工程的输出信号，这样小的外部输入信号产生了更加稳定的输出。注意当输入边带信号频率超过 2 kHz 时，将会产生混叠。

例 2.15　使用 8000 点的查找表产生扫频的正弦波

图 2.25 给出了使用 8000 点的查找表产生扫频正弦信号的程序 sweep8000.c，头文件 sine8000_table.h 含有 8000 个数据点，表示一个周期正弦波。因为输出点速率 F_s = 8 kHz，输出 8000 个数据点正好是 1 s 的时间，（辅助材料中的）头文件利用了 MATLAB 中的编程语句：

*1000*sin(2*pi*i*start_freq/8000)*

产生，图 2.26 给出了头文件 sine8000_table.h 的一部分。

```
//Sweep8000.c Sweep sinusoid using table with 8000 points

#include "sine8000_table.h"            //one cycle with 8000 points
short start_freq = 100;                //initial frequency
short stop_freq = 3500;                //maximum frequency
short step_freq = 200;                 //increment/step frequency
short amp = 30;                        //amplitude
short delay_msecs = 1000;              //# of msec at each frequency
short freq;
short t;
short i = 0;

void main()
{
 comm_poll();                          //init DSK, codec, McBSP
 while(1)                              //infinite loop
  {
   for(freq=start_freq;freq<=stop_freq;freq+=step_freq)
    {                                  //step thru freqs
     for(t=0; t<8*delay_msecs; t++)    //output 8*delay_msecs samples
      {                                // at each freq
       output_sample(amp*sine8000[i]); //output
       i = (i + freq) % 8000;          //next sample is + freq in table
      }
    }
  }
}
```

图 2.25　使用 8000 点查找表产生扫频正弦波的程序（sweep8000.c）

设定初始频率为 100 Hz，步进为 200 Hz，最高频率为 3500 Hz，产生的信号频率有 100 Hz，300 Hz, 500 Hz, ⋯, 3500 Hz，每一个频率信号持续时间都为 1 s。

将 delay_msecs 从 1000 增加到 2000，从而产生一个较慢的扫频速率，这样每个频率信号持续时间为 2 s。如果 step_freq 增加到 700，产生的信号频率就是 100 Hz, 800 Hz, 1500 Hz, 2200 Hz 和 2900 Hz。

索引变量以 freq 递增，它决定从表中选取的数据点（参见例 2.4），例如为了产生 100 Hz 频率的信号，表中每隔 100 个点选择一个点，输出 80 个数据点相当于一个周期，8000 点对应 100 个周期。使用这种方案，8000 点总是用于以每秒 x 周期的速率生成各个频率的信号。

建立并运行工程 sweep8000，检验输出是否为正弦扫频信号。注意源程序 sweep8000.c 不是中断驱动的（用合适的矢量文件），可用滑动条来控制变量 amp，进而控制频率信号的幅度，用 delay_msecs（扫描速率）变量控制每个频率的持续时间，用 step_freq 控制频率的步进。

```
//sine8000_table.h Sine table with 8000 points generated with MATLAB

short sine8000[8000]=
{0, 1, 2, 2, 3, 4, 5, 5,
6, 7, 8, 9, 9, 10, 11, 12,
13, 13, 14, 15, 16, 16, 17, 18,
19, 20, 20, 21, 22, 23, 24, 24,
25, 26, 27, 27, 28, 29, 30, 31,
31, 32, 33, 34, 35, 35, 36, 37,
38, 38, 39, 40, 41, 42, 42, 43,
44, 45, 46, 46, 47, 48, 49, 49,
50, 51, 52, 53, 53, 54, 55, 56,
57, 57, 58, 59, 60, 60, 61, 62,
63, 64, 64, 65, 66, 67, 67, 68,
69, 70, 71, 71, 72, 73, 74, 75,
75, 76, 77, 78, 78, 79, 80, 81,
82, 82, 83, 84, 85, 86, 86, 87,
88, 89, 89, 90, 91, 92, 93, 93,
94, 95, 96, 96, 97, 98, 99, 100,
100, 101, 102, 103, 103, 104, 105, 106,
107, 107, 108, 109, 110, 111, 111, 112,
.
.
.
-13, -12, -11, -10, -9, -9, -8, -7,
-6, -5, -5, -4, -3, -2, -2, -1};
```

图 2.26 8000 点的正弦信号部分数据（sine8000_table.h）

例 2.16 产生伪随机噪声序列

图 2.27 给出了产生伪随机噪声序列的程序 noise_gen.c，伪随机序列是基于最大长度序列技术的算法，采用软件方法产生。先给寄存器赋一个 16 位的初始种子值，位 b0，b1，b11 和 b13 相异或，并将异或的结果放入反馈变量中。放有初始种子的移位寄存器再左移一位，反馈变量再被赋给移位寄存器的 b0 位。根据寄存器 b0 位是 0 还是 1，给 prnseq 赋一个定标的最小值或最大值，该值对应于噪声信号的幅度。辅助材料中的头文件 noise_gen.h 定义了寄存器的各比特位。

建立并运行工程 noise_gen，可以看到时域内的噪声或听到噪声信号，将噪声电平按一定比例增加到±16 000，检查产生的噪声信号电平是否变大了。将信号输出连接到频谱分析仪上，检查输出频谱在截止频率近似为 3500 Hz 的频带内是否相对平坦，3500 Hz 是编解码器板上抗混叠滤波器的带宽。

```
//Noise_gen.c Pseudo-random sequence generation

#include "noise_gen.h"                   //header file for noise sequence
int fb;
shift_reg sreg;                           //shift reg structure

interrupt void c_int11()                  //interrupt service routine
{
 int prnseq;                              //for pseudo-random sequence

 if(sreg.bt.b0)                           //sequence{1,-1}based on bit b0
      prnseq = -8000;                     //scaled negative noise level
 else
      prnseq = 8000;                      //scaled positive noise level
 fb =(sreg.bt.b0)^(sreg.bt.b1);           //XOR bits 0,1
 fb ^=(sreg.bt.b11)^(sreg.bt.b13);        //with bits 11,13 ->fb
 sreg.regval<<=1;                         //shift register 1 bit to left
 sreg.bt.b0 = fb;                         //close feedback path

 output_sample(prnseq);                   //output scaled sequence
 return;                                  //return from interrupt
}

void main()
{
 sreg.regval = 0xFFFF;                    //set shift register
 fb = 1;                                  //initial feedback value
 comm_intr();                             //init DSK, codec, McBSP
 while (1);                               //infinite loop
}
```

图 2.27 伪随机噪声序列产生程序（noise_gen.c）

参考文献

1. *TLC320AD535C/I Data Manual Dual Channel Voice/Data Codec*, SLAS202A, Texas Instruments, Dallas, TX, 1999.
2. S. Norsworthy, R. Schreier, and G. Temes, *Delta–Sigma Data Converters: Theory, Design and Simulation*, IEEE Press, Piscataway, NJ, 1997.
3. P. M. Aziz, H. V. Sorensen, and J. Van Der Spiegel, An overview of sigma delta converters, *IEEE Signal Processing*, Jan. 1996.
4. J. C. Candy and G. C. Temes, eds., *Oversampling Delta–Sigma Data Converters: Theory, Design and Simulation*, IEEE Press, Piscataway, NJ, 1992.
5. C. W. Solomon, Switched-capacitor filters, *IEEE Spectrum*, June 1988.
6. *PCM3002/PCM3003 16-/20-Bit Single-Ended Analog Input/Output Stereo Audio Codecs*, SBAS079, Texas Instruments, Dallas, TX, 2000.
7. *TMS320C6000 McBSP: AC'97 Codec Interface*, SPRA528, Texas Instruments, Dallas, TX, 1999.

第 3 章　C6x 处理器的结构和指令系统

本章主要介绍 TMS320C6x 处理器的硬件结构和指令系统、寻址方式、汇编器指令、线性汇编器以及使用 C 语言、汇编和线性汇编指令编写的程序实例。

3.1　引言

1982 年，TI 公司推出了第一代 TMS32010 数字信号处理器，1986 年又接着推出了 TMS320C25 数字信号处理器[1]，1991 年推出了 TMS320C50 数字信号处理器。C1x，C2x 和 C5x 不同系列不同型号的处理器具有不同的特点，例如在处理速度方面就有很大的不同，但这些 16 位的处理器都是定点处理器且指令是兼容的。

在冯・诺依曼总线结构处理器中，程序指令和数据存储在同一个存储器空间里。采用冯・诺依曼总线结构的处理器，一个指令周期内只能进行一次读或写存储器操作，而典型的 DSP 应用需要在一个指令周期内访问存储器数次。定点处理器 C1x，C2x 和 C5x 基于改进的哈佛总线结构，程序指令和数据具有独立存储空间，因此允许同时访问数据和程序指令。

ADC 的量化误差和舍入舍出噪声是定点处理器所面临的问题。ADC 利用一个最佳的估计数值代替输入信号样值，例如，假设考虑一个 ADC 的字长为 8 位，输入信号范围是-1.5 V 到+1.5 V，则 ADC 的量化步长是：输入范围 $/2^8 = 3/256 = 11.72$ mV，因此量化过程产生的误差可能达到±11.72 mV/2 = ±5.86 mV。当输入电压不是 11.72 mV 的整数倍时，ADC 用一个最佳值来表示输入信号样值。对于 8 位的 ADC 来说，可用 2^8 或 256 个不同的电平来表示输入信号。对于字长更长的 ADC，例如 16 位的 ADC（这种器件现在比较常见），它能够减小量化误差，从而具有更高的分辨率。ADC 的字长位数越多，就能更精确地表示输入信号的大小。

TMS320C30 是 20 世纪 80 年代后期推出的浮点处理器，C31，C32 和最近的 C33 都属于 C3x 系列浮点处理器的成员[2,3]，随后推出的 C4x 系列浮点处理器与 C3x 系列在指令上是兼容的，它们都基于改进的哈佛总线结构[4]。

1997 年推出的 TMS320C6201（C62x）是 C6x 系列定点数字信号处理器的第一个成员，和以前的定点处理器 C1x，C2x 和 C5x 有所不同，C62x 采用了超长指令字体系结构（VLIW），程序指令和数据仍然使用和哈佛总线结构一致的独立存储空间。VLIW 结构具有简单的指令，但和传统结构的 DSP 相比，对于一个具体的应用来说，VLIW 处理器需要更多的指令。

C62x 和前几代的定点处理器指令是不兼容的，后来推出的 TMS320C6701（C67x）浮点处理器是 C6x 系列处理器的又一个成员，C62x 系列定点处理器的指令集是 C67x 处理器指令集的子集，附录 A 列出了 C6x 处理器的指令，C64x 是最近推出 C6x 系列的一个定点处理器。

一个专用集成电路芯片（ASIC）内含有一个 DSP 内核和特定应用相关的用户电路，C6x 处理器可以作为标准通用的 DSP，通过编程实现特定的应用。在调制解调器、回声抵销器及其他产品中都含有专用数字信号处理器。

定点处理器更适于使用电池的设备中，譬如蜂窝电话。因为与一个同等的浮点处理器相比，

它具有较小的功耗，C1x，C2x 和 C5x 是 16 位的定点处理器，具有有限的动态范围和精度，而 C6x 定点处理器是 32 位的处理器，具有较好的动态范围和精度。对于定点处理器，需要对数据进行定标。溢出也是一个需要注意的问题，这种情况发生在进行运算时，例如两个数相加，计算结果比处理器寄存器的位数更多，这时就会发生溢出现象。

浮点处理器的价格一般来说相对较高，因为处理器芯片有一些额外的电路，以进行整数和浮点算术处理，因而芯片具有更多的资源或体积相对更大一些。当选择专用数字信号处理器时，价格、功耗和运算速度是几个重要的考虑因素。C6x 系列处理器包括定点处理器（如 C62x、C64x）和浮点处理器（如 C67x），特别适合应用于需要较强计算能力的场合。当然也可以选择其他公司的数字信号处理器，如 Motorola，Analog Devices 等公司的产品[5]。

其他的结构，包括 Super Scalar 结构，需要专用硬件决定哪些指令并行执行。和 VLIW 结构处理器不一样，对于 Super Scalar 结构处理器，程序员的负担相对较小，而处理器的负担相对较大。它不一定执行相同的一组指令，因此难于定时，很少应用于 DSP 中。

3.2 TMS320C6x 的结构

DSK 板上的 TMS320C6711 是基于 VLIW 结构的浮点处理器[6-9]，内部存储器采用两级高速缓存结构，包括第一级 4 kB 的高速程序缓存（L1P）、4 kB 高速数据缓存（L1D）和第二级 64 kB RAM 或者数据/程序高速缓存（L2）。TMS320C6711 具有同步存储器（SDRAM 和 SBSRAM）和异步存储器（SRAM 和 EPROM）的直接接口。虽然同步存储器需要时钟信号，但是它是介于选择静态 SRAM 和动态 SDRAM 的比较折中的办法，因为 SRAM 的速度虽然比 DRAM 快，但是价格却相对较高。

DSP 芯片上的外围设备包括两个多通道串行口（McBSP）、两个定时器、一个 16 位主机端并行接口（HPI）和一个 32 位扩展存储器接口（EMIF）。DSP 外围 I/O 接口的电源电压是 3.3 V，而内核电压是 1.8 V。内部总线包括一个 32 位程序地址总线和一个 256 位程序数据总线。256 位程序数据总线可以提供存放 8 个 32 位指令或可分别作为两个 32 位的数据地址总线、两个 64 位的数据总线和两个 64 位的数据存储总线。通过 32 位的地址总线，DSP 寻址空间总共可以达到 2^{32} = 4 GB，包括 4 个外部存储器空间：CE0，CE1、CE2 和 CE3。图 3.1 是包含在 CCS 中 C6711 处理器的功能单元框图。

C6x 芯片上独立的存储体允许在一个指令周期内访问存储器两次，两个独立的存储体可以通过两套独立的总线进行访问。因为内部存储器是按照存储体来组织的，所以可以并行执行两条装载或者两条存储指令。只要同时访问不同的存储体，就不会产生数据存取冲突。独立的程序、数据以及直接存储器访问（DMA）总线使 C6x 可以并行执行指令读取、数据读写以及 DMA 操作。因为数据和指令存放在不同的存储空间，因此可以并行访问存储空间，进行程序指令的读取和数据的存取。C6x 还有可按字节寻址存储空间，内部存储器由相互独立的程序存储器和数据存储器组成，通过内部 32 位端口（C64x 为两个 64 位端口）访问内部存储空间。

DSK 上的 C6711 包含地址从 0x00000000 开始的 72 kB 内部存储器以及 16 MB 外部 SDRAM，通过 CE0 映射到从 0x80000000 开始的地址空间。除此之外，DSK 板上还有 128 kB 的闪存，其地址从 0x90000000 开始。图 3.2 是包含在 CCS 中的两级内部存储器结构框图。表 3.1 列出了存储器的映射地址空间。在 CCS 中，还有 DSK 的电路原理图，文件名为 C6711dsk_schematics.pdf。

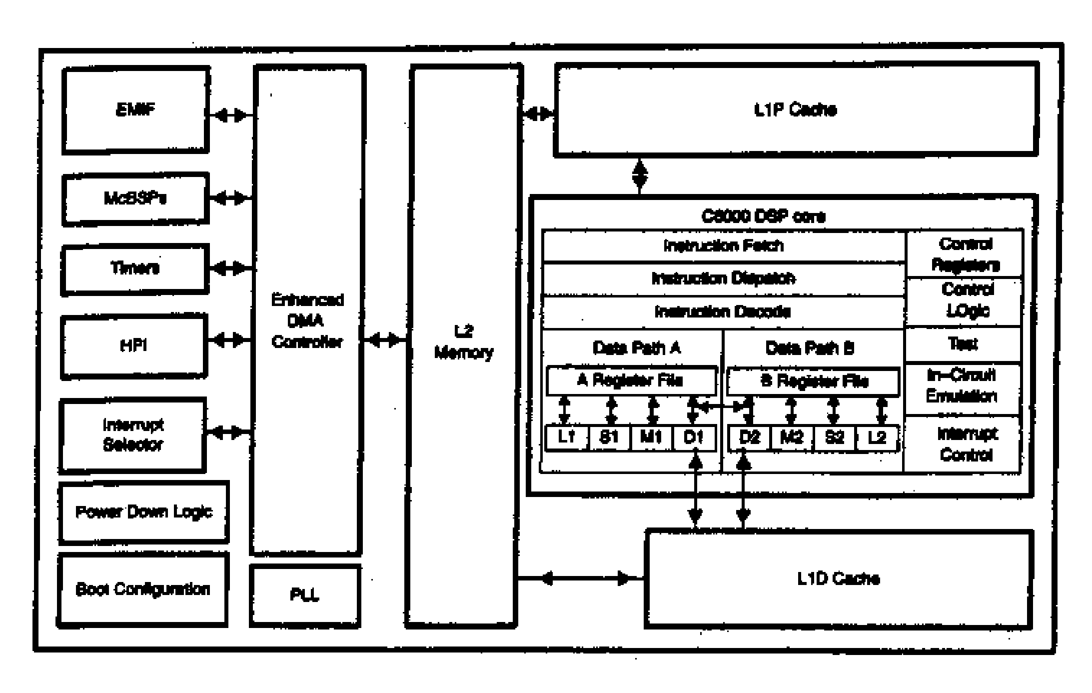


图 3.1 TMS320C6x 的功能单元框图

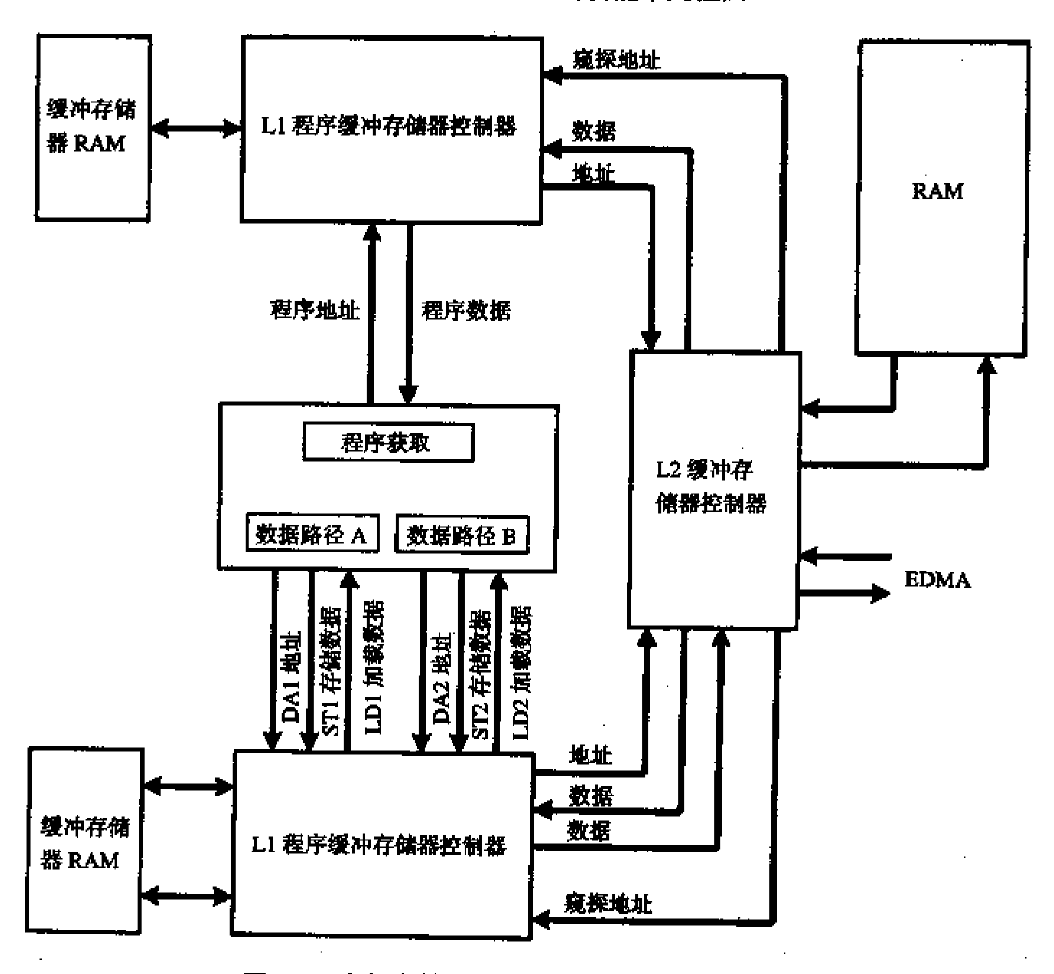


图 3.2 内部存储器结构框图（由 TI 公司提供）

表 3.1　存储器地址空间分配情况

地址范围（Hex）	大小	内存块说明
0000 0000—0000 FFFF	64 KB	内部 RAM（L2）
0001 0000—017F FFFF	24 MB–64 KB	保留
0180 0000—0183 FFFF	256 KB	内部配置总线 EMIF 寄存器
0184 0000—0187 FFFF	256 KB	内部配置总线 L2 控制寄存器
0188 0000—018B FFFF	256 KB	内部配置总线 HPI 寄存器
018C 0000—018F FFFF	256 KB	内部配置总线 McBSP 0 寄存器
0190 0000—0193 FFFF	256 KB	内部配置总线 McBSP 1 寄存器
0194 0000—0197 FFFF	256 KB	内部配置总线定时器 0 寄存器
0198 0000—019B FFFF	256 KB	内部配置总线定时器 1 寄存器
019C 0000—019F FFFF	256 KB	内部配置总线中断选择器寄存器
01A0 0000—01A3 FFFF	256 KB	内部配置总线 EDMA RAM 和寄存器
01A4 0000—01FF FFFF	6 MB–256 KB	保留
0200 0000—0200 0033	52	QDMA 寄存器
0200 0034—2FFF FFFF	736 MB–52	保留
3000 0000—3FFF FFFF	256 MB	McBSP 0/1 数据
4000 0000—7FFF FFFF	1 GB	保留
8000 0000—8FFF FFFF	256 MB	外部内存接口 CE0
9000 0000—9FFF FFFF	256 MB	外部内存接口 CE1
A000 0000—AFFF FFFF	256 MB	外部内存接口 CE2
B000 0000—BFFF FFFF	256 MB	外部内存接口 CE3
C000 0000—FFFF FFFF	1 GB	保留

DSK 板上如果采用 150 MHz 的时钟，理想情况下一个周期内可以实现两次乘法和加法运算，每秒时间内总共可实现 3 亿次乘法和累加操作（MAC）。图 3.1 画出了 C6x 的 8 个功能单元，使用其中能够处理浮点操作的 8 个功能单元（不包括下面将要介绍的.D 功能单元），每秒可执行 9 亿次浮点操作。也就是说，DSP 工作在 150 MHz 时钟时，处理速度相当于执行每秒 12 亿条指令，其指令周期为 6.67 ns。

3.3　功能单元

如图 3.1 所示，CPU 包括 8 个独立的功能单元，分成 A 和 B 两组独立的数据通道。每个通道含有一个乘法操作单元（.M），一个逻辑和算术运算单元（.L），一个分支、位操作和算术运算单元（.S）以及装载/存储和算术单元（.D），其中.S 和.L 单元用来执行算术、逻辑以及分支指令，所有数据传输都使用.D 单元。

算术运算，如加法或减法（SUB 或 ADD），可以由除.M 单元以外的所有功能单元执行（每个数据通道有一个.M 单元）。8 个功能单元包括 4 个浮点/定点 ALU（两个.L 单元和两个.S 单元）、两个定点 ALU（.D 单元）以及两个浮点/定点乘法器（.M 单元）。每个功能单元都可以在自己的通道内直接读或写寄存器组中的寄存器，每个通道含有 16 个 32 位的寄存器，A 通道含有 A0~A15

的 16 个寄存器，B 通道含有 B0~B15 的 16 个寄存器。以 1 结尾的单元写入寄存器组 A，以 2 结尾的单元写入寄存器组 B。

两个交叉通道（1x，2x）可以使一个数据通道中的功能单元访问在另一个数据通道寄存器组中的一个 32 位的操作数，在每个周期内有一个交叉通道数据读取的最大次数。一个数据通道中的每个功能单元可利用一个交叉通道访问另一个数据通道中的寄存器数据（也就是说，一个通道中的功能单元能够访问另一个数据通道中的寄存器），共有 32 个通用寄存器，但其中有些寄存器用于特定寻找方式或条件指令。

3.4 取指和执行包

TI 公司推出的 VELOCITI 结构源于 VLIW 结构，一个执行包（EP）由一组在相同时钟周期时间内能并行执行的指令组成。在一个取址包（FP）内，EP 的个数可以从 1 个（EP 内有 8 条并行指令）到 8 个（EP 内没有并行执行指令，每个执行包只有一个指令）。改进的 VLIW 结构允许一个 EP 内包含多个 EP。

每个 32 位指令的最低有效位用来确定下一条或后续的指令是否属于同一个 EP（当其为 1 时）或是下一个 EP 的一部分（当其为 0 时）。假设一个 FP 有三个 EP：EP1（有两条并行的指令）、EP2 和 EP3，后两个都有三条并行的指令，三个 EP 分别如下所示：

```
        Instruction A
||      Instruction B

        Instruction C
||      Instruction D
||      Instruction E

        Instruction F
||      Instruction G
||      Instruction H
```

EP1 包含两个并行指令 A 和 B，EP2 包含三条并行指令 C，D 和 E，EP3 包含三条并行指令 F, G 和 H。FP 如图 3.3 所示，每个 32 位指令的第 0 比特位（也就是最低有效位）是位 p，该比特位用来表示这条指令是否和后续指令并行执行。举一个例子，假设指令 B 的 p 位是 0，表明它和后继指令 C 不在相同的 EP 内。同样地，指令 E 和指令 F 也不在相同的 EP 内。

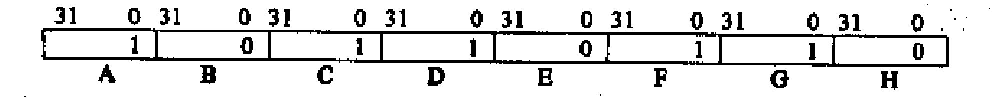


图 3.3 含有三个执行包的 FP 及其与 p 位关系

3.5 流水线技术

流水线操作技术是数字信号处理器的重要特点，它可以使指令正确地并行工作，但需要仔细考虑定时问题。流水线操作过程包括三个状态周期：取指令周期、指令译码周期和指令执行周期。

1. 取指令周期包括 4 个时钟周期：

(a) PG，（在 CPU 中）产生取指的程序地址。

(b) PS，程序地址送到程序存储器地址线上。

(c) PW，程序地址准备好，等待读程序存储器。

(d) PR，（CPU 中的）程序取指包从程序存储器读取操作码。

2. 指令译码周期包括两个时钟周期：

(a) DP，将一个 FP 中的指令分别分发到合适的功能单元。

(b) DC，指令译码。

3. 执行周期包含的时钟周期和指令及处理器有关，执行周期时间从 6 个时钟周期（对于定点处理器）到 10 个时钟周期（对于浮点处理器），延迟时间和执行的指令有关：

(a) 乘法指令（Multiply），占用两个时钟周期，其中一个时钟周期是延迟时间。

(b) 赋值指令（Load），占用 5 个时钟周期，其中 4 个时钟周期是延迟时间。

(c) 分支指令（Branch），占用 6 个时钟周期，其中 5 个时钟周期是延迟时间。

表 3.2 给出了流水线操作的三个状态周期，表 3.3 给出了流水线操作的过程以及结果。表 3.3 的第一行表示时钟周期 1, 2, …, 12，下面的每一行代表一个 FP。行中用 PG, PS, …表示每个取指周期 FP 的不同时钟周期，第一个 FP 的 PG 时钟周期从第一个时钟周期开始，第二个 FP 的 PG 时钟周期从第二个时钟周期开始，依次类推。每个 FP 用 4 个时钟周期取指令，两个时钟周期指令译码，但指令执行状态周期可能占用 1 到 10 个时钟周期（表 3.3 没有给出执行状态周期里全部的时钟周期），我们假定每一个 FP 包含一个 EP。

例如，在第 7 个时钟周期，第一个 FP 中的指令在指令执行周期第一个时钟周期（指令执行状态周期可以只有一个时钟周期），第二个 FP 的指令处于指令译码状态时钟周期，第三个 FP 的指令处于指令分发时钟周期等，所有的 7 个指令都运行在不同的状态周期，因此在第 7 个时钟周期，流水线是“满负荷”的，也就是说，从第 7 个时钟周期开始，流水线上时钟有 7 条指令在并行执行着。

表 3.2　流水线各个状态周期

程序获取				解　码		执　行
PG	PS	PW	PR	DP	DC	E1-E6（E1-E10 用于双精度）

表 3.3　流水线操作过程和结果

时钟周期											
1	2	3	4	5	6	7	8	9	10	11	12
PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
			PG	PS	PW	PR	DP	DC	E1	E2	E3
				PG	PS	PW	PR	DP	DC	E1	E2
					PG	PS	PW	PR	DP	DC	E1
						PG	PS	PW	PR	DP	DC

大部分指令执行周期只有一个时钟周期，有些指令占用较多的时钟周期，如乘法指令（MPY），数据传输指令（LDH/LDW）和分支指令（B）分别占用2个、5个和6个时钟周期。指令执行需要的额外时钟周期与指令的浮点和双精度类型操作是有关的，有时占用时间达到10个时钟周期。例如，在C67x上实现双精度乘法操作（MPYDP），需要9个延迟时钟周期，因此整个执行状态周期需要10个时钟周期。

功能单元的等待时间表示一条指令绑定某个功能单元时所占用的时钟周期数，除了C67x具有的双精度指令外，所有指令的功能单元延迟时间都是一个时钟周期。功能单元的等待时间和延迟时钟周期是不同的，例如MPYDP指令有4个功能单元的等待时间，却需要9个时钟周期，这意味着其他指令在4个时钟周期内不能使用相关的功能单元。一个数据保存指令没有延迟时钟周期，但在流水线操作的第三个时钟周期才结束执行。

如果乘法指令的结果（例如MPY指令）被下一条指令中用到，为了使流水线正常工作，就必须在MPY之后插入NOP（空操作）指令。当指令中用到一个数据传输或分支指令的结果时，分别需要插入4个或5个NOP指令。

3.6 寄存器

共有两个寄存器组，每组有16个可访问的寄存器，A组寄存器包括A0~A15，B组寄存器包括B0~B15。寄存器A0，A1，B0，B1，B2是条件寄存器，寄存器A4~A7，B4~B7是用于循环寻址的寄存器，寄存器A0~A9和B0~B9（除B3外）用做临时寄存器。A10~A15以及B10~B15的任何寄存器用于从子程序返回时数据的保存和恢复，类似于堆栈操作时所用到的存储器。

一个40位的数据可保存在寄存器对中，低32位存放在偶寄存器中（例如A2），剩下的8位存储在相邻的高（奇）地址寄存器中（A3）。对64位双精度数据，也可以采用同样的方法，即分别用奇偶寄存器对来存储数据。

这32个寄存器可以当做通用寄存器，一些特殊功能寄存器还可用于控制和中断，如附录B所示，地址模式寄存器（AMR）用循环寻址和中断控制寄存器。

3.7 线性和循环寻址方式

寻址模式决定了访问存储器的方式，规定了如何访问数据，例如间接地从存储单元获得一个操作数。TMS320C6x支持线性和循环寻址方式，最常用的寻址方式是存储器间接寻址方式。

3.7.1 间接寻址

间接寻址方式可以和有偏移及无偏移寻址方式一起使用，寄存器R表示A0~A15和B0~B15的32个寄存器中的任何一个寄存器，用寄存器指定或指向一个存储器单元地址，因此这些寄存器内容就是地址指针。间接寻址通过在32个寄存器中的任何一个寄存器前面加上星号来表示。为了说明问题，假设R是一个地址寄存器，下面不同的表示方法具有不同的意义：

1. *R。表示寄存器R中存放的是存储某个数据的存储单元地址。
2. *R++(*d*)。表示寄存器R中存放存储单元的地址指针，当访问完寄存器R指针对应的存储单元后，R寄存器的指针就改变，寄存器R内新指针值是原来的指针值加上偏移量*d*。如果$d = 1$（默认值），新的指针值就是$R + 1$，即R的值就变为相邻的高地址。如果用两

个减号（——）代替前面的两个加号（++），结果正好相反，即当访问完寄存器 R 指针对应的存储单元后，寄存器 R 内新指针值是原来的指针值减去偏移量 d，R 的值变为 $R-d$。

3. *++R(d)。表示地址指针先加上偏移量 d，因此当前的地址指针是 $R+d$。如果使用两个减号（——），则先减去当前的偏移量，这时当前的地址指针变为 $R-d$。
4. *+R(d)。表示地址指针先加上 d，因此当前的地址指针是 $R+d$（这和前面的表示方法是一样的）。但在这种情况下，R 先加偏移量后仅用于寻址，但是 R 的内容本身并不改变，这和前面的表示方法是不同的。

3.7.2　循环寻址

循环寻址用于创建环形缓冲区，这种缓冲区通过硬件方法来实现，在一些 DSP 算法中这种方法是非常有用的。例如在数字滤波或相关运算中，数据需要不断更新，这时利用环形缓冲区是一种比较好的方法，第 4 章给出了一个利用环形缓冲区更新"延迟"抽样、实现数字滤波的例子。

C6x 采用专门的硬件实现循环寻址方式，这种寻址方式可以和环形缓冲区一起使用，通过移动数据实现抽样值的更新，而消除了没有环形缓冲区直接移动数据所产生的开销（Overhead）。当指针到达环形缓冲区的末端或底部位置时，即指针指到缓冲区最后一个数据时，这时指针就会增大，且自动转回到或指向缓冲区的开始位置或顶部，即缓冲区中的第一个数据位置。

如附录 B 所示，在地址模式寄存器（AMR）中有两个独立的环形缓冲区，通过使用 BK0 和 BK1 来实现。A4~A7，B4~B7 的 8 个寄存器和两个.D 功能单元可用做指针（所有寄存器可用于线性寻址）。下面的程序代码段介绍了如何借助寄存器 B2（只有 B 组寄存器能够使用）使用环形缓冲区并在 AMR 内设定合适的值：

```
MVK    .S2  0x0004,B2  ;lower 16 bits to B2. Select A5 as pointer
MVKLH  .S2  0x0005,B2  ;upper 16 bits to B2. Select B0, set N = 5
MVC    .S2  B2,AMR     ;move 32 bits of B2 to AMR
```

两条数据搬移指令 MVK 和 MVKLH（使用.S 单元）将 0x0004 送到寄存器 B2 的低 16 位，将 0x0005 送到寄存器 B2 的高 16 位。MVC（常数搬移）指令是惟一一条能够访问 AMR 和其他控制寄存器（参见附录 B）的指令，并且只能与 B 组的功能单元和寄存器在 B 组寄存器上进行操作。首先给 B2 赋一个 32 位的数据，然后通过 MVC 指令访问 AMR，将该数据传送到 AMR 中[6]。

数据 0x0004 =$(0100)_b$ 赋给 AMR 寄存器的低 16 位，即 AMR 寄存器的位 2（也就是第三位）设置为 1，其他比特位都设为 0，这样就将模式设置为 01，寄存器 A5 作为指向环形缓冲区的指针，而 BK0 块作为环形缓冲区。

表 3.4 给出了与寄存器 A4~A7 和 B4~B7 的相关模式。0x0005 =$(0101)_b$ 赋给 AMR 的高 16 位，即将 AMR 寄存器的位 16 和位 18 设置为 1，其他比特位设置为 0，该数对应缓冲区的大小，用 BK0 作为环形缓冲区，由于高 16 位对应的数 $N=5$，因此缓冲区的容量为 $2^{N+1}=64$ 字节。例如，如果想用 BK0 得到一个容量为 128 字节的缓冲区，AMR 的高 16 位应设置为$(0110)_b$ = 0x0006。如果使用汇编程序对环形缓冲区进行操作，当该汇编程序段执行完成后，程序返回到一个 C 调用函数，这时 AMR 寄存器就要被重新初始化为默认的线性模式，因此为了进行正确的运算，必须保存指针的地址。

表 3.4 AMR 的可用模式及其作用

模　式	说　明
00	用于线性寻址（默认为复位）
01	用于周期寻址，使用 BK0
10	用于周期寻址，使用 BK1
11	保留

3.8 TMS320C6x 指令集

3.8.1 汇编语句格式

汇编语句格式可用下面的结构表示：

```
Label  ||  [ ]  Instruction  Unit  Operands  ;comments
```

如果有标号的话，它表示一个特殊的地址或者是包含指令或数据的存储单元地址，标号必须从第一列开始。并行双竖线表示当前指令是否和前面的指令并行执行。后面的[ ]字段是可选项，它使相关指令是有条件的，而 A1，A2，B0，B1 和 B2 可用做条件寄存器。例如，[A2]表示如果[A2]不为 0，则执行相关指令。相反使用[!A2]，表示如果 A2 为 0 时，则执行相关的指令。所有 C6x 的指令都可用 A1，A2，B0，B1 和 B2 作为条件寄存器，通过判断它是否为 0 决定是否执行相关指令。指令段可以是汇编器指令或者是汇编指令的助记符，汇编器指令就是汇编器的命令，例如.word value 表示保留一个 32 位的存储单元，在该存储单元内存储指定的数据 value。助记符是 DSP 工作时一条真正执行的指令，助记符或汇编器指令不能从第一列开始。Unit 段是可选的，它可以是 CPU 内 8 个功能单元中的任一个单元。当注释语句从第一列开始时，语句前面可以用分号或星号，而从其他列开始的注释语句，必须在注释语句前面用分号。

浮点处理器 C3x/C4x 的指令和定点处理器 C1x，C2x，C5x/C54x 的指令是不兼容的，但定点处理器 C62x 的指令与浮点处理器 C67x 的指令是兼容的。C62x 指令实际上是 C67x 指令的子集，只有 C67x 处理器才有处理双精度和浮点操作的附加指令（一些附加指令在定点处理器 C64x 上也有）。

为了说明 C6x 指令集，书中给出了一些程序代码段。C6x 处理器汇编指令和 C3x/C4x 的指令很相似，C62x/C67x 的单任务指令使其比以前的定点或浮点处理器更容易编程，这应归功于高效编译器。C64x 的附加指令（C62x 上没有的）和 C3x/C4x 处理器的多任务类型指令相似。阅读书中所讨论程序的注释部分对理解程序是大有帮助的，附录 B 给出了 C62x/C67x 的指令集。

3.8.2 指令类型

下面说明汇编程序的语法规则，尽管 8 个功能单元指定字段是可选的，但在程序调试和提高程序效率、优化程序方面，还是非常有用的。关于这一点，将在第 8 章中进行讨论。

1．加法、减法和乘法指令

（a）`ADD  .L1  A3,A7,A7` ;add A3 + A7 →A7 (accum in A7)

将寄存器 A3 和 A7 中的内容相加，并将结果放到寄存器 A7 中。功能单元.L1 是可选项，如果结果放在寄存器 B7 中，则使用功能单元.L2。

(b) `SUB .S1 A1,1,A1 ;subtract 1 from A1`

使用.S 功能单元，将寄存器 A1 内容减去 1，结果存放到寄存器 A1 中，实现递减。

(c)

```
    MPY   .M2  A7,B7,B6  ;multiply 16LSBs of A7,B7 → B6
||  MPYH  .M1  A7,B7,A6  ;multiply 16MSBs of A7,B7 → A6
```

这是在同一个指令包中的两条并行执行指令，第一条指令是寄存器 A7 和 B7 的最低 16 位相乘并将结果存在寄存器 B6 中，第二条指令将寄存器 A7 和 B7 的高 16 位相乘，结果存到寄存器 A6 中。在这种情况下，两个乘法/加法操作可在单个指令周期内完成，这可用于将一组积的和分解成两组积的和：一组用低 16 位对 1, 3, 5, …进行运算，另一组用高 16 位对 2, 4, 6, …进行运算，注意并行指令标识符号“||”不能放在第一列上。

2. 数据传送指令

(a)

```
    LDH  .D2  *B2++,B7   ;load (B2) →B7, increment B2
||  LDH  .D1  *A2++,A7   ;load (A2) →A7, increment A2
```

第一条指令将寄存器 B2 所指定的存储单元的半字（16 位）存放到 B7，然后 B2 的内容加 1，指向下一个高地址单元；同样另一个间接寻址的并行指令将寄存器 A2 所指定的存储单元的内容传送到寄存器 A7，寄存器 A2 的内容再加 1，指向相邻的下一个高地址单元。

指令 LDW 可以传送一个 32 位的字。使用 LDW 指令以及功能单元.D1 和.D2，可将存储器中的数据传送到寄存器 A 和寄存器 B 中。C6711 的双字浮点传送指令 LDDW 可将两个 32 位的寄存器数据传送到 A 组寄存器，同时将两个 32 位的寄存器数据传送到 B 组寄存器。

(b) `STW .D2 A1,*+A4[20] ;store A1→(A4) offset by 20`

将寄存器 A1 中的 32 位字存储到地址为寄存器 A4 所指定且偏移 20 字（32 位的字）或 80 字节的存储单元。地址寄存器 A4 先加上偏移值，但是寄存器的内容并不改变（如果改变 A4 的值，在寄存器前面使用两个加号）。

3. 分支/数据移动指令

下面的程序指令段说明了分支和数据移动指令的用法：

```
Loop MVK   .S1  x,A4      ;move 16LSBs of x address →A4
     MVKH  .S1  x,A4      ;move 16MSBs of x address →A4
     .
     .
     .
     SUB   .S1  A1,1,A1   ;decrement A1
[A1] B     .S2  Loop      ;branch to Loop if A1 # 0
     NOP        5         ;five no-operation instructions
     STW   .D1  A3,*A7    ;store A3 into (A7)
```

第一条指令将地址为 x 的存储单元低 16 位（LSB）传送到寄存器 A4，第二条指令将地址为 x 的存储单元高 16 位（MSB）传送到寄存器 A4。寄存器 A4 现在存有地址为 x 单元的全部 32 位数据。为了给寄存器赋一个 32 位的常数，必须使用 MVK/MVKH 指令。

寄存器 A1 用做循环计数器。随着 SUB 指令的执行，其数值逐步递减，使用该数值

来判断是否需要转移。如果 A1 不为 0，则执行 loop 到该句的循环体，否则（即 A1 等于 0），继续执行循环体外程序指令段，将寄存器 A3 中的数据传送到 A7 指定地址的存储单元中。

3.9 汇编器指令

汇编器指令就是给汇编器（而不是编译器）的信息，它并不是程序指令，而只是在汇编过程中被汇编器执行的指令，所以不像指令那样占据存储器空间。汇编器指令不会产生可执行代码，它可用来指定不同程序段的地址。例如，汇编器指令.sect “my_buffer” 定义了名为 my_buffer 的数据或程序代码段，指令.text 和.data 分别表示文本段和数据段定义指令，其他汇编指令如.ref 和.def 分别用来表示未定义符号和定义符号。汇编器通过汇编指令来建立不同的段，例如通过.text 建立程序代码段，.bss 用于建立全局变量或静态变量。

其他常用的汇编器指令有：

1. .short，定义 16 位整型数。
2. .int，定义 32 位整型数（还有.word 或.long）。编译器认为长整型数是 40 位的，而 C6x 编译器长整型数是 32 位的。
3. .float，定义 32 位的 IEEE 单精度常数。
4. .double，定义 64 位的 IEEE 双精度常数。

通过汇编器指令.byte，.short 或.int 可给变量赋初始值，使用指令.usect 可以定义未初始化的段（如.bss 段），而指令.sect 建立初始化的段。例如，命令.usect “variable”，128，2 分别指定名为 variable 的未初始化段，段的大小以字节为单位并按字节进行对齐。

3.10 线性汇编

除了 C 语言和汇编程序外，还有一种方法就是线性汇编。汇编优化器代替 C 编译器，它和线性汇编源程序（扩展名为.sa）结合起来生成汇编源程序（扩展名是.asm），这与 C 编译优化器和 C 源程序代码相结合生成汇编程序在很大程度上是相同的，但用汇编优化器生成的汇编程序比用 C 编译优化程序生成的汇编程序效率更高，由 C 语言源程序和线性汇编产生的汇编源程序必须经过汇编后才能生成目标代码。

线性汇编编程方法在编程所需时间和程序代码效率两者之间实现了较好的折中，汇编优化器确定使用哪些功能单元和寄存器（也可以由用户来指定），查找可并行执行的指令，并进行软件流水线操作优化（第 8 章将讨论该问题）。本章的结尾部分介绍了两个编程实例，用来说明用 C 程序如何调用一个线性汇编函数。在线性汇编程序中，并行指令是无效的。和汇编程序一样，在线性汇编程序中可以选择指定使用哪些功能单元。

经过多年的发展，C 编译优化器的效率已经越来越高。虽然 C 语言程序比汇编语言程序（在执行速度方面）效率低，但它比汇编程序更省时省力，易于开发。对 C 语言程序也可用手动优化以达到百分之百的效率，但编程比较费时。

值得注意的一个有趣现象是：C6x 的汇编程序语法并不像 C2x/C5x 或 C3x 系列数字信号处理器那样复杂。实际上，使用汇编语言对 C6x 进行编程是比较简单的。例如，C3x 指令：

```
DBNZD  AR4,LOOP
```

第一个 D 表示循环计数器 AR4 递减，第二个 D 表示延迟，如果 AR4 不为 0，分支（B）转移到 LOOP 指定的地址。由于采用流水线技术，具有延迟的分支指令可在一个时钟周期内高效地执行，虽然 C6x 没有这种多任务指令（虽然最近 C64x 有这种指令），但是实际上，C6x 的指令更"简单"。例如，有单独的指令可使计数器递减（用 SUB 指令）和转移，这些简单的指令适合于用高效的 C 编译器进行编译。

虽然使用汇编语言进行编程，实现预期的任务是非常简单的，但这并不意味着很容易就能编写或把程序转变成高效的汇编语言程序。手动优化生成完全高效的汇编语言程序是非常有意义的，但相对来说比较困难。

线性汇编语言是汇编语言编程和 C 语言编程结合的中间产物，它使用汇编指令语法：如 ADD，SUB，MPY，但操作数/寄存器和 C 语言中使用的一样。在某些情况下，使用线性汇编是在选择 C 语言和汇编语言编程之间的折中。

线性汇编器指令包括：

```
.cproc
.endproc
```

它们用来指定一个 C 可调用程序或汇编优化器优化的一段代码，另一个指令.reg 用于声明变量和定义保存在寄存器中的数据名。本章后面部分将用程序实例说明如何用 C 调用汇编语言函数或 C 调用线性汇编函数。

3.11　在 C 程序中使用汇编语句

汇编指令和命令可以嵌入在 C 程序中，并用 asm 语句来声明。asm 语句可以访问硬件，而只用 C 语言是很难直接访问硬件的。使用 asm 语句的语法是：

```
asm ("assembly code");
```

在引号内的每行汇编程序语句应该是有效的汇编语句。注意：如果指令有标号，标号的第一个字母必须跟在第一个引号后面，这样可使标号从第一列开始。汇编语句必须是合法有效的语句，因为编译器并不检查它的语法错误，而是将它直接复制到编译的输出文件里。如果汇编语句有语法错误的话，汇编器将会查出来。

应尽量避免在 C 程序中使用 asm 语句，特别是线性汇编程序中。这是因为汇编优化器可能在 asm 语句附近重新安排程序语句的位置，这样就可能导致意想不到的错误。

3.12　C 可调用汇编函数

本章后面举了两个例子，说明用 C 程序如何调用汇编函数。寄存器 B3 保留下来，用于保存调用函数的返回地址。

在 C 程序中调用汇编函数，可用 extern 语句作为函数的外部声明，如：

```
extern int func();
```

汇编函数 func 返回一个整数值，该语句是可选的。

3.13 定时器

DSP内有两个32位的定时器，可以用来定时、计数或中断CPU，还能控制外部ADC开始转换或控制DMA控制器进行数据传送。定时器包括时钟周期寄存器、定时计数寄存器和定时控制寄存器，时钟周期寄存器用来确定定时器工作的时钟频率，定时计数寄存器用来保存计数值，定时控制寄存器用来监视定时器的状态。

3.14 中断

中断可以由内部或外部触发，发生中断后将暂时停止CPU当前的进程，而转入执行中断所触发的特定任务，程序指针转向中断服务程序，中断源可以是ADC、定时器等。在执行中断服务程序前，必须保存当前的进程状态，以便中断服务程序执行完后返回主程序时，恢复进入中断服务程序前的进程状态，即保存相关寄存器的状态，继续执行中断服务程序，当中断服务程序执行完后，再恢复原先寄存器的状态。

共有16个中断源，包括两个定时器中断、4个外部中断、4个McBSP中断以及4个DMA中断。有12个CPU中断可供使用，它们通过中断选择器进行选择。

3.14.1 中断控制寄存器

中断控制寄存器（参见附录B）如下：

1. CSR（控制状态寄存器），包含全局中断使能位（GIE）和其他控制/状态位。
2. IER（中断启用寄存器），启用/禁止各个中断。
3. IFR（中断标志寄存器），显示中断状态。
4. ISR（中断设置寄存器），设置中断悬挂。
5. ICR（中断清除寄存器），清除中断悬挂。
6. ISTP（中断服务表指针），定位ISR。
7. IRP（中断返回指针）。
8. NRP（非屏蔽中断返回指针）。

中断具有优先级。复位（Reset）具有最高优先级，复位和非屏蔽中断（NMI）分别具有第一优先级和第二优先级，它们都由外部引脚进行控制。中断使能寄存器用于设置特定的中断，中断标志寄存器可用于检查是否有中断发生以及发生了哪种中断。

和复位一样，NMI是非屏蔽中断，但NMI可通过CSR中的NMIE位使其屏蔽（禁止）。只有在复位或非屏蔽中断时，它才被设为0。如果NMIE设置为0，则INT4~INT15的所有中断都被禁止，附录B介绍了各种中断寄存器。

复位信号是低电平有效信号，它用于停止CPU的执行，而NMI信号用于告警，通知CPU有潜在的硬件问题。12个低优先级的CPU中断是可用的，对应于可屏蔽的中断信号引脚：INT4~INT15，其中INT4的优先级最高，INT15的优先级最低。为了能产生非屏蔽中断，非屏蔽中断必须使位NMIE为1（高电平有效）；当复位（或在NMI被先置位后）时，NMIE位被清成0，这时就可以产生复位中断。

为了能够执行可屏蔽中断，控制状态寄存器（CSR）中的全局中断启用位（GIE）和中断启

用寄存器（IER）中的 NMIE 位需要置 1。CSR 的位 0、GIE 位以及 IER 的位 1 和 NMIE 位都设置为 1。注意，可以利用 CSR 和–2 相与（即使用二进制补码表示，除了最低有效位为 0 外，其他位全为 1）来设置 CSR 的 GIE 位为 0，从而实现全局屏蔽掉所有的可屏蔽中断。

对于需要的可屏蔽中断，应该将对应的中断启用位（IE）也设置为 1。当中断发生时，相应的中断标志寄存器（IFR）的标志位被置为 1，以表示中断状态的发生。为了能产生一个可屏蔽中断，需要按以下步骤进行设置：

1. GIE 位置 1。
2. NMIE 位置 1。
3. 相应的 IE 位置 1。
4. 相应 IFR 的位置 1。

对于一个发生的中断，CPU 一定不在执行相关的分支指令的延迟时隙中。

中断服务表（IST）如表 3.5 所示，该表表示中断的起始单元地址，每个地址单元对应一个中断的取指包（FP）。表中共有 16 个 FP，每个 FP 有 8 条指令。右边的地址对应于每个指定中断的偏移量。例如，对于中断 INT11，FP 的地址是基址加偏移量 160h。因为每个 FP 包含 8 个 32 位的指令（即 256 位）或 32 字节，所以表中每个偏移地址是以 20h = 32 递增的。

表 3.5 中断服务表

中 断	偏 移
RESET	000h
NMI	020h
保留	040h
保留	060h
INT4	080h
INT5	0A0h
INT6	0C0h
INT7	0E0h
INT8	100h
INT9	120h
INT10	140h
INT11	160h
INT12	180h
INT13	1A0h
INT14	1C0h
INT15	1E0h

来源：TI 公司。

复位的 FP 必须从 0 地址单元开始，但其他中断的 FP 可以重新定位。重新定位地址可通过将地址写到中断服务表指针（ISTP）寄存器的中断服务表基地址（ISTB）寄存器，如图 B.7 所示。复位时，ISTB 为 0。当重新确定矢量表的位置时，需要使用 ISTP，重新定位地址是 ISTB

加上偏移量。

表 3.6 给出了选择指定类型中断时，中断选择器对应设定的值。中断选择器设定为 01000 时，也用于选择增强的 DMA 中断（EDMA_INT）。

表 3.6 利用中断选择器选择中断

中断选择器	类　型	说　明
00000	DSPINT	主端口到 DSP 中断
00001	TINT0	定时器 0 中断
00010	TINT1	定时器 1 中断
00011	SD_INT	EMIF SDRAM 定时器中断
00100	EXT_INT4	外部中断引脚 4
00101	EXT_INT5	外部中断引脚 5
00110	EXT_INT6	外部中断引脚 6
00111	EXT_INT7	外部中断引脚 7
01000	DMA_INT0	DMA 通道 0 中断
01001	DMA_INT1	DMA 通道 1 中断
01010	DMA_INT2	DMA 通道 2 中断
01011	DMA_INT3	DMA 通道 3 中断
01100	XINT0	McBSP0 发送中断
01101	RINT0	McBSP0 接收中断
01110	XINT1	McBSP1 发送中断
01111	RINT1	McBSP1 接收中断

软件定义的中断 INT4~INT15 和相应的物理中断信号相对应，通过中断复用寄存器 IML 和 IMH 定义中断。为了实现 INT4~INT15 的相应中断，表 3.5 所示的中断选择值应该存放在合适的 IML 或 IMH 段中[7]。关于这一点，也可参见支持文件 C6xdskinterrupt.h。

3.14.2 XINT0 的选择

在前面的大部分例子中，都选择了 MsBSP0 传输中断。在通信文件 C6xdskinit.c 中，调用了函数 Config_Interrupt_Selector，该函数在中断头文件 C6xinterrupts.h 中。从 C6xinterrupts.h 中得到相应的中断选择值（01100）= 0xC（该 5 位的选择值对应于 IMH 寄存器的第 5 位到第 9 位）。

3.14.3 中断响应

IACK 和 INUMx（INUM0~INUM3）是 C6x 引脚上的信号，表示中断已经产生和正在执行。4 个 INUMx 信号用来表示正在处理哪个中断，例如，INUM3 = 1 (MSB)，INUM2 = 0，INUM1 = 1，INUM0 = 1 (LSB)，对应于（1011）$_b$ = 11，表示正在执行中断 INT11。

IE11 设置为 1 以使能 INT11，通过读取中断标志寄存器可以确认 IF11 位是否被置为 1（INT11 被启用），从而校验中断是否正在执行。将中断设置寄存器（ISR）的某位置 1，会使 IFR 中相应的中断标志位置位；而将中断清除寄存器（ICR）的某位置 0，则会使 IFR 中相应的中断标志位清除。

当 CPU 执行到分支指令的等待延迟时隙时，所有的中断将保持悬挂状态。因为分支指令有 5

个延迟时隙，因此小于 6 个时钟周期的中断信号是不会产生中断的。只要没有等待执行的分支指令，CPU 就会处理悬挂的中断，文献[6]对此有更多的介绍。

3.15　多通道缓冲串行口

C6x 有两个多通道缓冲串行口，它们可以和廉价的（符合工业标准的）外设接口。MsBSP 具有全双工通信、收发独立时钟和成帧以及直接与 AC97 和 IIS 兼容设备接口等特点，允许 8~32 位数据进行传送。McBSP 有关的输入输出时钟和成帧参数技术指标可参见文献[7]。

当内部数据在传输时，外部数据也可以进行通信，图 3.4 给出了 McBSP 的内部原理图。数据发送（DX）和接收（DR）引脚用于数据通信，控制信号（时钟信号和帧同步信号）通过 CLKX，CLKR，FSX 和 FSR 引脚来实现。CPU 或 DMA 控制器从数据接收寄存器（DRR）读入数据，把将发送的数据写到数据发送寄存器（DXR），发送移位寄存器把数据移位到 DX 上，接收移位寄存器把 DR 上接收到的数据复制到接收缓冲寄存器，然后，接收缓冲寄存器上的数据再复制到接收移位寄存器，进而被 CPU 或 DMA 控制器读取。

其他寄存器包括：串行口控制寄存器（SPCR）、接收/发送控制寄存器（RCR/XCR）、接收/发送通道启用寄存器（RCER/XCER）、密码控制寄存器（PCR）以及抽样速率产生寄存器（SRGR），这些寄存器支持更多的数据通信[7]。

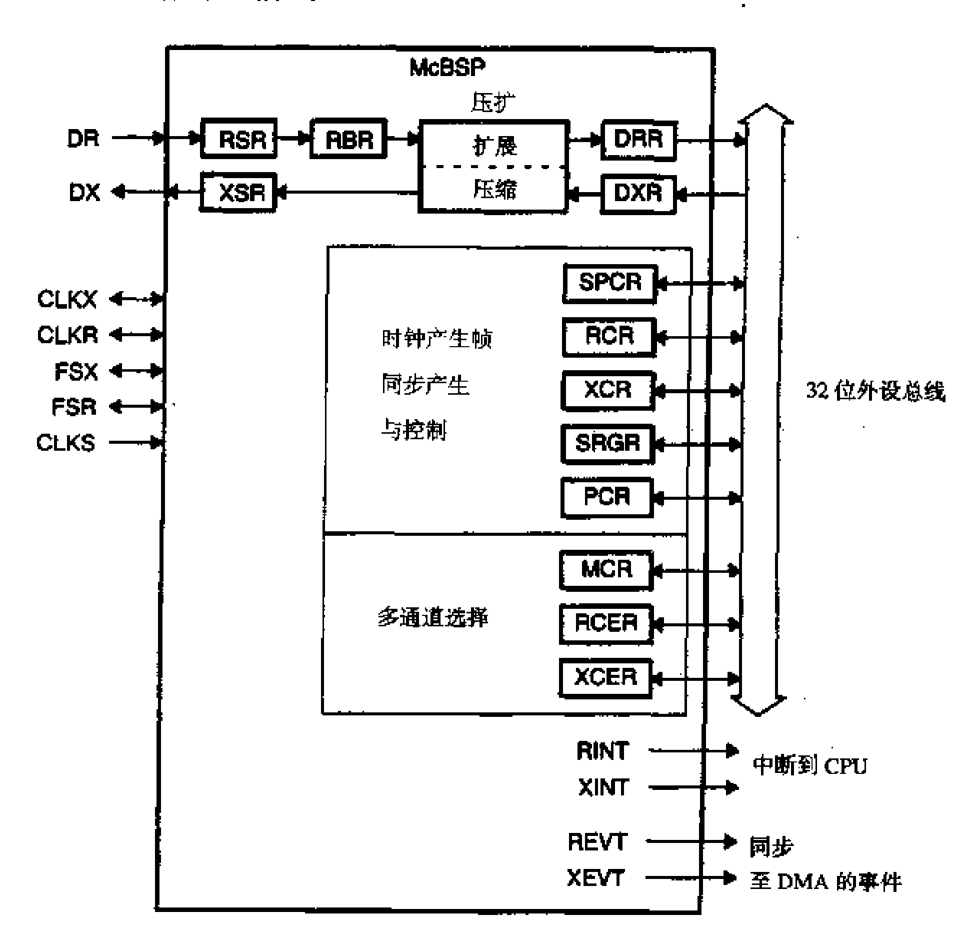


图 3.4　McBSP 内部原理图

3.16 直接存储器存取方式

直接存储器存取方式（DMA）允许内存和外设之间直接进行数据传输而无需 CPU 的干预，4 个 DMA 通道可以独立配置进行数据传送。除此之外，还有一个附加（辅助）的通道用于主机端口接口（HPI）的 DMA。DMA 可直接访问片内存储器和外部存储器接口（EMIF），可以传输不同长度的数据，包括 8 位的字节、16 位的半字和 32 位的字。

有许多 DMA 寄存器用于配置 DMA 工作方式：地址寄存器（源地址和目的地址）、索引寄存器、计数重载寄存器、DMA 全局数据寄存器和控制寄存器。源地址和目的地址可能来自内部程序存储器、内部数据存储器、外部存储器接口和内部的外设总线，外部的引脚以及内部外设所产生的中断都可以启动 DMA 传输。

每个 DMA 通道可以用 CPU 编程具有不同的优先级。在 4 个 DMA 通道中，通道 0 具有最高优先级，而通道 3 的优先级最低。每个 DMA 通道都可以独立启动数据块的传输。一个数据块可以包含多个帧，一帧可包含多个单元，一个单元有一个独立的数据。DMA 计数重载寄存器内的数据包含两部分：表示帧数（高 16 位）的部分和表示单元数（低 16 位）的部分。一些 DSP 有增强 DMA（EDMA）通道，有多达 16 个独立的可编程通道。

3.17 存储数据需要考虑的问题

3.17.1 数据分配

在连接命令文件中，可以对程序代码和数据段在存储器区域中位置进行分配。这些段可以事先赋给初始值，也可不赋初始值。除.text 段外，这些已赋初始值或未赋初始值的段不能分配到内部程序存储器中。

赋初始值的段有：

1. .cinit，定义全局静态变量段。
2. .const，定义全局静态常量段。
3. .switch，包含大型开关语句的跳转表。
4. .text，定义可执行程序代码和常量段。

未赋初始值的段有：

1. .bss，定义全局静态变量段。
2. .far，定义远程全局静态变量段。
3. .stack，分配系统堆栈存储器。
4. .sysmem，malloc，calloc，realloc 分配函数的动态存储空间。

连接程序可用于确定将不同段放在哪些地方更为合理，例如，为了最高效地执行程序，可以将码段放在内部快速存储器中。

3.17.2 数据存取格式

C6x 总是对按一定方式排列的数据进行访问，可按字节、半字和字（32 位）寻址。数据格式包括 4 字节、两个半字和字方式，如 32 位数据传送语句用 LDW，地址是按字对齐的，所以地址

的低两位是 0，否则就会发生数据传送错误。C6x 还能访问双字数据（64 位）。.S1 和.S2 两个功能单元都可用于执行双字指令 LDDW，进行两个 64 位双字数据的传送，从而实现每时钟周期传送 128 位数据。

3.17.3 Pragma 命令

Pragma 命令通知编译器注意几个函数，这些命令包括 DATA_ALIGN，DATA_SECTION 等。DATA_ALIGN 的 pragma 命令的语法是：

```
#pragma DATA_ALIGN (symbol,constant);
```

该语句指定 symbol 的对齐方式，constant 是以 2 为底的幂。在后面的 FFT 例子中，使用 pragma 命令将数据按一定方式存放在存储器中。

DATA_SECTION 的 pragma 命令的语法是：

```
#pragma DATA_SECTION (symbol,"my_section");
```

该语句为 symbol 在名为 my_section 的段中分配存储空间。

另一个有用的 pragma 命令是：

```
# pragma MUST_ITERATE (20,20)
```

该语句通知编译器下面循环执行 20 次（最大最小值都是 20）。

3.17.4 存储器模式

在默认情况下，编译器按小存储器模式生成程序代码，如果数据不声明为远程（far）型，数据都被默认为近程（near）型。如果使用 DATA_SECTION 的 pragma 指令，数据对象就被声明为 far 远程变量。

如何调用实时运行支持函数可通过两种选项来进行控制，第一种是利用选项-mr0 及实时支持数据和 near 调用进行控制，第二种是利用选项-mr1 及实时支持数据和 far 调用进行控制。使用 far 方式调用函数并不表示这些函数一定驻留在片外存储器中。

大存储器模式可用连接器选项-ml$x$（x 为 0~4 的数字）来实现。如果没有说明，在默认情况下，数据和函数都被认为是 near 型。只有当调用的函数偏移 1 M 字以上的情况下，这时才会用大存储器模式。

3.18 定点和浮点格式

在附录 C 中，我们回顾了有关定点的一些问题。

3.18.1 数据类型

数据类型包括以下 4 种。

1. short：16 位短整型。用二进制补码表示时，表示范围为$-2^{15}\sim2^{15}-1$。
2. int 或 signed int：32 位整型。用二进制补码表示时，表示范围为$-2^{31}\sim2^{31}-1$。
3. float：32 位浮点型。用 IEEE 32 位表示时，表示范围是 $2^{-126} = 1.175\,494 \times 10^{-38}$ 到 $2^{+128} = 3.402\,823\,46 \times 10^{38}$。

4. double：64 位双精度型。用 IEEE 64 位表示时，其范围是 $2^{-1022}=2.225\ 073\ 85\times10^{-308}$ 到 $2^{+1024}\approx1.797\ 693\ 13\times10^{+308}$。

数据类型对程序性能有一定的影响，如 short 型定点乘法比 int 型更有效（执行周期少），使用 const 同样可以提高程序性能。

3.18.2 浮点格式

浮点处理器有较宽的动态范围，定标是比较容易的。如图 3.5 所示，浮点数据可用 32 位的单精度数或者用 64 位的双精度数来表示。如图3.5(a)所示，在单精度数表示格式中，位 31 表示符号位，位 23 到位 30 是指数位，位 0 到位 22 是小数位。小到 10^{-38} 的数和大到 10^{+38} 的数都可用单精度数据格式来表示。在双精度表示格式中，如图3.5(b)所示，指数位和小数位有更多的位数。因为使用 64 位表示数据，所以需要使用寄存器对。第一个寄存器的位 0 到位 31 代表小数位，第 2 个寄存器位 0 到位 19 也是小数位，位 20 到位 30 是指数位，位 31 表示符号位，因此使用双精度格式可以表示小到 10^{-308} 和大到 10^{+308} 的数据。

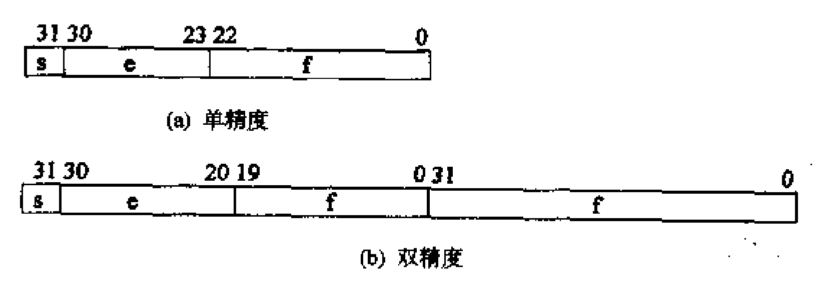


图 3.5 数据格式

以 SP 和 DP 结尾的指令分别表示单精度和双精度指令，有些浮点指令比定点指令延迟时间长。例如：定点乘法 MPY 需要 1 个延迟或 NOP，而单精度 MPYSP 指令需要 3 个延迟，双精度指令 MPYDP 则达到 9 个延迟。

单精度浮点指令 ADDSP 和 MPYSP 有 3 个延迟时隙，需要 4 个时钟周期完成指令的执行；双精度指令 ADDDP 和 MPYDP 分别有 6 和 9 个延迟时隙。浮点双字传输指令 LDDW（与定点指令 LDW 一样，具有 4 个延迟时隙）可传送 64 位，两个 LDDW 指令可通过.S1 和.S2 并行执行，每个周期传送 128 位。

一个单精度浮点数可以存到一个寄存器里，而一个 64 位的双精度浮点数需要存到寄存器对中，如 A1:A0, A3:A2, …, B1:B0, B3:B2, …，最低 32 位有效位保存到偶数的寄存器对中，而最高 32 位有效位存放到奇数的寄存器对中。

当使用浮点指令时，必须权衡动态范围、精度和运行速度的影响。

3.18.3 除法

浮点 C6711 处理器有单精度倒数指令 RCPSP，除法运算可以通过利用分母的倒数乘以分子来实现[6]。尽管没有定点的除法指令，但有些程序可用来进行除法运算，这些程序利用定点处理器实现 Newton-Raphson 算法，达到除法运算的目的。

3.19　程序改进

第 8 章讨论了几种程序优化方法，包括使用定点、浮点实现及汇编程序的优化方法。

3.19.1　内部函数

C 程序可利用运行库支持文件里的许多内部函数进行进一步优化，内部函数类似于运行支持库函数。内部函数可进行乘法、加法、求平方根的倒数运算等。例如：使用内部函数_mpy 代替星号算子进行乘法运算。内部函数是一些特殊的函数，可直接映射插入到 C6x 的指令中。例如：

```
int _mpy()
```

等效于汇编指令 MPY，它将两个数的最低 16 位相乘。内部函数 `int _mpyh()` 等效于汇编指令 MPYH，将两个数的最高 16 位相乘。

3.19.2　循环计数的 trip 指令

线性汇编指令.trip 是用于指定循环迭代的次数，如果事先知道和使用确定的次数，线性汇编优化器可产生流水线程序（将在第 8 章中讨论），而不会产生多余的循环，这种方法可同时缩短程序的长度和执行时间。即使不是确切的数值，例如当实际迭代次数是某指定值的倍数时，.trip 指令也可以用来改进性能。内部函数_nassert( )在 C 程序中可替代.trip 指令，例 3.1 在点积的程序例子中说明了_nassert( )的使用方法。

3.19.3　交叉路径

数据和地址交叉路径指令用于提高程序代码的效率，指令：

```
MPY  .M1x  A2,B2,A4
```

说明了数据路径交叉过程，两个数 A2 和 B2，分别来自 A 和 B 寄存器组，相乘的结果放在 A4 中。如果结果存放在 B 寄存器组中，就要在指令中使用 2x 交叉路径，这时指令变成：

```
MPY  .M2x  A2,B2,B4
```

乘的结果存放在 B4 中。指令：

```
LDW  .D1T2  *A2,B2
```

表示一个地址交叉路径，将寄存器 A2（A 组寄存器）的内容传送到寄存器 B2（B 组寄存器）中。在 C6x 中，只有两个可用的交叉路径，所以在一个周期内，不能有两个以上的指令使用交叉路径。

3.19.4　软件流水线

软件流水线就是使用可用的资源得到高效的流水线程序代码，它的目标是在一个循环体内使用所有的 8 个功能单元，但是，使用软件流水线方法需要花很多编程时间。获得流水线程序需要三个步骤：

1. 开始部分
2. 循环内核（循环体）
3. 结尾部分

开始部分包含建立第二步循环体的指令，结尾部分（最后一步）包含完成所有循环迭代的指令。当优化选择级别采用-o2 或-o3 时，编译器就会使用软件流水线方法。最有效的软件流水线程序代码有循环过程计数器，该计数器是递减的，例如：

```
for(i=N; i !=0;i--)
```

举一个点积的例子，有两个数组，每个数组有 N 个数，数的宽度为 32 位，用手工编写的流水线程序代码需要 $N/2+8$ 个周期计算两个数组积的和。按这种方式计算，计算 200 个数的数组积的和，如第 8 章所述，就需要 108 个时钟周期。这种效率是通过使用一些指令来实现的，如用 LDW 指令传送 32 位字，用 mpy 和 mpyh 两条指令分别实现高 16 位和低 16 位的乘法。

去掉结尾部分可以减小程序代码的长度，可用选择项-ms$n$（$n=0,1,2$）通知编译器，优先考虑程序代码长度的优化，然后再对性能进行优化。为了手工编写软件流水线程序，先要画一张关联关系图，同时建立一个时间进度表[8]。在第 8 章中，我们将讨论与程序代码效率有关的软件流水线方法。

3.20 约束因素

3.20.1 存储器约束

内部存储器由多个块组成，可同时对数据进行读取和存储。因为每个存储器块是单端口，因此每个周期只能访问一次某个存储块。如果一个周期内访问不同的存储块，那么一个周期内可访问不同的存储块可达两次。如果多次访问同样的存储块（在同一空间内），那么流水线操作就会停止，从而导致完成执行需要附加的时钟周期。

3.20.2 交叉路径约束

因为两条数据路径的每边只有一个交叉路径，所以每个周期至多有两条指令使用交叉路径。由于使用了两个可用的交叉路径，下面的程序语句是有效的：

```
    ADD  .L1x  A1,B1,A0
||  MPY  .M2x  A2,B2,B3
```

但下面的程序语句就是无效的，因为两条指令使用同一条交叉路径：

```
    ADD  .L1x  A1,B1,A0
||  MPY  .M1x  A2,B2,A3
```

与功能单元连在一起的 x 表示一个交叉路径。

3.20.3 读取/存储约束

使用的地址寄存器必须和.D 单元位于相同的数据路径，下面的程序语句是有效的：

```
    LDW  .D1  *A1,A2
||  LDW  .D2  *B1,B2
```

而下面这两条语句却是无效的：

```
    LDW  .D1  *A1,A2
||  LDW  .D2  *A3,B2
```

另外，读取和存储不能是在同一个寄存器组，如果一条读取（或存储）指令使用一个寄存器组，另一条并行执行的读取（或存储）指令必须使用另一个寄存器组。例如：

```
    LDW  .D1  *A0,B1
||  STW  .D2  A1,*B2
```

下面这两条指令也是有效的：

```
    LDW  .D1  *A0,B1
||  LDW  .D2  *B2,A1
```

但是下面这两条并行执行指令就是无效的：

```
    LDW  .D1  *A0,A1
||  STW  .D2  A2,*B2
```

3.20.4　在一个取指包内多个执行包对流水线的影响

在表 3.3 中，给出了在一个取指包内并行执行的 8 条指令流水线操作的过程。表 3.7 给出了在一个取指包内，当有多个执行包时流水线操作的过程。

考虑 6 个取指包（FP1~FP6）的流水线操作，FP1 包含三个执行包，其余每个取指包包含一个执行包。从第 2 个时钟周期到第 5 个时钟周期，FP2 到 FP6 的每个取指包启动各自的程序取指周期，当 CPU 检测到 FP1 含有多个 EP 时，它强制流水线暂停工作，以致 FP1 的 EP2 和 EP3 分别在第 6 个和第 7 个时钟周期启动相应的分发过程。在一个 FP 内的每条指令有一个 p 比特位，该位说明这条指令是否和后续的指令并行执行（如果该位是 1，表明这条指令和后续指令是并行执行的，如图 3.3 所示）。

表 3.7　流水线的暂停效应

时钟周期											
1	2	3	4	5	6	7	8	9	10	11	12
PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
					DP	DC	E1	E2	E3	E4	E5
						DP	DC	E1	E2	E3	E4
	PG	PS	PW	PR	X	X	DP	DC	E1	E2	E3
		PG	PS	PW	X	X	PR	DP	DC	E1	E2
			PG	PS	X	X	PW	PR	DP	DC	E1
				PG	X	X	PS	PW	PR	DP	DC
					X	X	PG	PS	PW	PR	DP

从第 1 个到第 4 个时钟周期，程序开始相应的取指周期。在一个取指包中的三个 EP 会导致流水线暂停工作，这就使 EP2 在第 6 个时钟周期（不是第 5 个时钟周期）、EP3 在第 7 个时钟周期启动 DP 过程。后面只有一个 EP（8 条并行指令）的取指包 FP2 暂停下来，这样前一个取指包（FP1）的三个 EP 都通过分发 DP 过程。因此，尽管 FP2 在第二个时钟周期开始取指，但它的分发 DP 过程需要延迟到第 8 个时钟周期才能开始。第三取指包（FP3）也只有一个执行包，在第三个时钟周期开始取指，由于流水线暂停操作，它的 DP 分发直到第 9 个时钟周期才开始。

因此，一个 FP 内有三个 EP，流水线操作需要暂停两个时钟周期，表 3.7 描述了流水线暂停操作的情况。表中流水线暂停发生在第一个 FP，它有 4 个 EP，每个 EP 有两个并行指令。

3.21 TMS320C64x 处理器

C6000 系列的另一个成员是 C64x，C64x 可在千兆赫兹的较高时钟速率下工作。在 750 MHz 时钟速率下，每个时钟周期 8 条指令，相当于 6000 MIPS（每秒 6 000 000 000 条指令）。

C64x 基于 VELOCITI.2 结构，是 VELOCITI 结构的扩展[8]。它的一些特点包括：具有大存储器空间，是多数存储器容量的两倍，共有 64 个 32 位的寄存器。额外的寄存器适用于分组数据类型的操作，支持 4 个 8 位或两个 16 位或一个 32 位的寄存器进行运算，因此增强了并行处理的能力。例如，MPYU4 指令在一个指令周期时间内执行 4 个 8 位的乘法。另外，还添加了一些特殊的指令，这些指令用于无线通信和数字图像处理中遇到的许多运算，在这些应用中，8 位数据处理是经常的事情。除此之外，.M 单元（乘法运算的功能单元）也能执行移位和循环移位操作。同样，.D 单元（数据处理功能单元）也能执行逻辑运算。

C64x 是定点处理器，现有的指令适合于更多的单元，双字读取（LDDW）和存储（STDW）指令每个周期可访问 64 位的数据，还有两个双字读取和存储指令（每个周期可访问 128 位的数据）。

C64x 处理器添加了许多指令。例如，指令：

```
BDEC   LOOP,B0
```

使计数器 B0 递减，并且（基于 B0）执行到 LOOP 语句的条件分支。在计数器递减之前进行分支条件判断，判断是根据 B0 是不是负数（并不是判断 B0 是否为 0）。这种多任务指令好像 C3x 和 C4x 系列处理器中的指令。

另外，利用内部 C 函数_dotp2，可实现两个 16 × 16 位的积，然后把积加起来，从而进一步减小运算所需要的时钟周期。这个内部 C 函数相应的汇编函数为 DOTP2，它使用两个乘法单元，每个时钟周期可实现 4 个 16 × 16 位数据的乘法，相当于 C62x 或 C67x 速率的两倍。当时钟速率是 750 MHz 时，相当于每秒 30 亿乘法运算，或者是每秒 60 亿次 8 × 8 运算。

3.22 程序范例

本节讨论 6 个程序例子，第一个是使用内部函数_nassert 提高点积运算效率的例子，其他 5 个例子介绍了汇编和线性汇编程序的实现：一个用 C 程序调用汇编函数、一个用 C 程序调用线性汇编函数以及一个汇编程序调用汇编函数。在这里，重点是介绍汇编和线性汇编程序的语法，并不是要求生成优化的程序。第 8 章将进一步讨论程序优化方法以及与之相关的程序效率和软件流水线方法。

例 3.1　高效点积程序

本例介绍使用内部函数_nassert 在点积运算中的优点，仍然使用第 1 章介绍的点积运算作为例子。图 3.6 给出了程序 dotpopt.c，它调用了图 3.7 中的 C 函数 dotpfunc.c。利用_nassert 调整输入数据指针，并将它作为常量指针，该函数生成了更高效率的程序代码，同时这给编译器提供了关于循环的附加信息。

检验一下：如果编译器选项选择-g 和-o3，函数 dotpfunc.c 的执行时间将从 100 个时钟周期（不用内部函数）降到 71 个时钟周期（使用内部函数）。如果使用编译器选项选择-g，-pm 和-o3，执行时间进一步减少到 30 个时钟周期。选项-pm 是进行程序级优化，此时源文件先被编译成中间文件，为了比较编译选项的影响，可以把该结果和用例 1.3 中的函数 dotp 得到的结果进行比较。

```
//dotpopt.c Optimized dot product of two arrays

#include    <stdio.h>
#include    "dotp4.h"
#define     count 4

short x[count] = {x_array};                 //declare 1st array
short y[count] = {y_array};                 //declare 2nd array
volatile int result = 0;                    //result

main()
{
 result = dotpfunc(x,y,count);              //call optimized function
 printf("result = %d decimal \n", result);  //print result
}
```

图 3.6 求点积的 C 主程序（dotpopt.c）

```
//dotpfunc.c Optimized dot product function

int dotpfunc(const short *a, const short *b, int ncount)
{
     int sum = 0;
     int i;

   _nassert((int)(a)%4 == 0);
   _nassert((int)(b)%4 == 0);
   _nassert((int)(ncount)%4 == 0);

   for ( i = 0; i < ncount; i++)
   {
       sum += (a[i] * b[i]);            //sum of products
   }
   return (sum);                        //return sum as result
}
```

图 3.7 使用_nassert 内部函数，被 C 程序调用求点积的函数（dotpfunc.c）

在第 8 章中，我们对两个数组的点积程序进行优化，其中每个数组有 N 个数。我们得到这样的结果：利用定点实现，执行时间将减少到 $7+N/2+1$ 时钟周期。或者说，当每个数组有 200 个数据时，执行时间就是 108 个时钟周期；而利用浮点实现时，执行时间就是 124 个时钟周期（见表 8.4）。

例 3.2 利用 C 调用汇编函数，求 $n+(n-1)+(n-2)+\cdots+1$ 的值

该例说明了如何利用 C 程序调用汇编函数。C 源程序 sum.c（如图 3.8 所示）调用汇编函数 sumfunc.asm（如图 3.9 所示），该程序实现求 $n+(n-1)+(n-2)+\cdots+1$ 的值。在 C 主程序中设置 n 的值，（按常规）该值传递给寄存器 A4，例如，多个数的地址可由寄存器 A4, B4, A6, …传递到汇编函数，汇编函数得到的求和结果返回给 C 程序的 result 变量，然后输出最后结果。

汇编函数的命名（按常规）通常以下划线开始。在函数 asm 中，寄存器 A4 里存放的数 n 传送到 A1 寄存器，并将 A1 寄存器设为循环计数器，A1 随着程序执行递减。程序代码的循环部分从标号或地址 LOOP 开始，在第一个分支语句 B 结束。第一次加法运算计算出 $n+(n-1)$ 的

值，结果放在 A4 中，A1 寄存器的值再递减到 $(n-2)$ 。分支语句是基于 A1 寄存器值的条件转移语句（只有 A1，A2，B0，B1，B2 可用做条件寄存器）。因为 A1 不为 0 时，循环分支语句返回到 LOOP 标号处，继续执行循环内的指令，这样寄存器的值 $A4 = n + (n-1)$ 再加上 $A1 = (n-2)$ ，循环一直执行到 $A1 = 0$ 时结束。

```
//Sum.c Finds n+(n-1)+...+1. Calls assembly function sumfunc.asm

#include <stdio.h>

main()
{
 short n=6;                    //set value
 short result;                 //result from asm function

 result = sumfunc(n);          //call assembly function sumfunc
 printf("sum = %d", result);   //print result from asm function
}
```

图 3.8 调用汇编程序求 $n + (n-1) + (n-2) + \cdots + 1$ 的 C 程序（sum.c）

```
;Sumfunc.asm Assembly function to find n+(n-1)+...+1

           .def       _sumfunc  ;function called from C
_sumfunc:  MV    .L1  A4,A1     ;setup n as loop counter
           SUB   .S1  A1,1,A1   ;decrement n

LOOP:      ADD   .L1  A4,A1,A4  ;accumulate in A4
           SUB   .S1  A1,1,A1   ;decrement loop counter
    [A1]   B     .S2  LOOP      ;branch to LOOP if A1#0
           NOP        5         ;five NOPs for delay slots
           B     .S2  B3        ;return to calling routine
           NOP        5         ;five NOPs for delay slots
           .end
```

图 3.9 工程 sum 中，被 C 程序调用的汇编函数（sumfunc.asm）

第二个分支指令（按常规）是 C 调用程序的返回地址 B3，最后的求和结果或累加的值保存到 A4 中，最终传送给 C 程序的 result 变量，5 个 NOP 指令（空操作）占用 5 个延迟时隙，加入这 5 个延迟时隙表示分支指令的延迟。

程序中选择了.S 和.L 功能单元，虽然不是必需的，但这样做是有用的，有利于调试和分析哪些功能单元能提高程序效率。同样，标号 LOOP 后的两个冒号和函数也不是必需的。

建立和运行名为 sum 的工程，将 C 程序中 n 设为 6，检验求和的结果及打印的值是否为 21。

例 3.3 用 C 程序调用汇编函数计算一个数的阶乘

该例是求 $n \leqslant 7$ 的数的阶乘 $n! = n(n-1)(n-2)\cdots(1)$ ，并进一步说明汇编程序的语法。与例 3.2 很相似，C 语言源程序 factorial.c 如图 3.10 所示，在该程序中设定 n 的值，它调用如图 3.11 所示的汇编函数 factfunc.asm。注释部分对理解程序是有帮助的。

寄存器 A1 仍然设为循环计数器，在从标号 LOOP 开始的循环内，第一个乘法运算是 $n(n-1)$ ，并在 A4 中累加。通过寄存器 A4，n 的初始值被传送给汇编函数 factfunc.asm。MPY 指令延迟一

个时隙，因此在其后面加一个 NOP 指令，循环程序一直执行到 A1 = 0 时结束。注意：在该程序中没有指定功能单元，最后阶乘的结果通过 A4 返回到 C 程序。

建立和运行名为 factorial 的工程，检验 factorial 变量的结果和输出的值 5040（7!）。注意：n 的最大值是 7，因为 8! 大于 2^{15}。

```
//Factorial.c Finds factorial of n. Calls function factfunc.asm

#include <stdio.h>                    //for print statement

void main()
{
 short n=7;                           //set value
 short result;                        //result from asm function

 result = factfunc(n);                //call assembly function factfunc
 printf("factorial = %d", result);    //print result from asm function
}
```

图 3.10　调用汇编函数求一个数阶乘的 C 程序（factorial.c）

```
;Factfunc.asm Assembly function called from C to find factorial

            .def   _factfunc          ;asm function called from C
_factfunc:  MV     A4,A1              ;setup loop count in A1
            SUB    A1,1,A1            ;decrement loop count
LOOP:       MPY    A4,A1,A4           ;accumulate in A4
            NOP                       ;for 1 delay slot with MPY
            SUB    A1,1,A1            ;decrement for next multiply
     [A1]   B      LOOP               ;branch to LOOP if A1 # 0
            NOP    5                  ;five NOPs for delay slots
            B      B3                 ;return to calling routine
            NOP    5                  ;five NOPs for delay slots
            .end
```

图 3.11　由 C 调用求一个数阶乘的汇编函数（factfunc.asm）

例 3.4　使用汇编程序调用汇编函数求数组的点积

该例求两个数组的点积，每个数组有 4 个数字，参见例 1.3，该例只使用了 C 语言程序，而例 3.2 和例 3.3 则介绍了汇编程序语法。图 3.12 是 dotp4a_init.asm 汇编程序清单，该程序用于初始化两个数组，并调用求两个数组点积的汇编函数 dotp4afunc.asm（如图 3.13 所示），同时该程序也通过寄存器 B3 设置返回地址并将结果地址送给寄存器 A0。两个数组的地址和数组的维数分别通过寄存器 A4、A6 和 B4 传送给汇编程序 dotp4afunc.asm，被调函数的结果通过 A4 返回，点积最后的结果存到地址为 result_addr 的存储器中。指令 STW 将保存在 A4 中的点积结果传送到 A0 做指针的存储器，这里寄存器 A0 作为地址 result_addr 的地址指针。

调用汇编程序的起始地址定义为 init，矢量文件 vectors_dotp4a.asm（如图 3.14 所示）定义了指向那个入口地址的分支语句，被调汇编函数 dotp4afunc.asm 计算积的和。因为寄存器 A6 不能作为条件寄存器（只有 A1，A2，B0，B1 和 B2 可作为条件寄存器），循环计数值赋给寄存器 A1。两个 LDH 指令（16 位半字）分别把两个数组的地址 x_addr 和 y_addr 赋给寄存器 A2 和 B2。例如，指令：

```
;Dotp4a_init.asm ASM program to init variables. Calls dotp4afunc.asm

              .def          init          ;starting address
              .ref          dotp4afunc    ;called ASM function
              .text                       ;section for code
x_addr        .short        1,2,3,4       ;numbers in x array
y_addr        .short        0,2,4,6       ;numbers in y array
result_addr   .short        0             ;initialize sum of products

init          MVK    result_addr,A4       ;A4 = lower 16-bit addr -->A4
              MVKH   result_addr,A4       ;A4 = higher 16-bit addr-->A4
              MVK    0,A3                 ;A3 = 0
              STH    A3,*A4               ;init result to 0
              MVK    x_addr,A4            ;A4 = 16 MSBs address of x
              MVK    y_addr,B4            ;B4 = 16 LSBs address of y
              MVKH   y_addr,B4            ;B4 = 16 MSBs address of y
              MVK    4,A6                 ;A6 = size of array
              B      dotp4afunc           ;branch to function dotp4afunc
              MVK    ret_addr,b3          ;B3 = return addr from dotp4a
              MVKH   ret_addr,b3          ;B3 = return addr from dotp4a
              NOP    3                    ;3 more delay slots(branch)

ret_addr      MVK    result_addr,A0       ;A0 = 16 LSBs result_addr
              MVKH   result_addr,A0       ;A0 = 16 MSBs result_addr
              STW    A4,*A0               ;store result
wait          B      wait                 ;wait here
              NOP    5                    ;delay slots for branch
```

图 3.12 调用求点积汇编函数的汇编程序（dotp4a_init.asm）

```
;Dotp4afunc.asm Multiply two arrays. Called from dotp4a_init.asm
;A4=x address,B4=y address,A6=count(size of array),B3=return address

              .def          dotp4afunc    ;dot product function
              .text                       ;text section
dotp4afunc    MV            A6,A1         ;move loop count -->A1
              ZERO          A7            ;init A7 for accumulation

loop          LDH           *A4++,A2      ;A2=(x. A4 as address pointer
              LDH           *B4++,B2      ;B2=(y). B4 as address pointer
              NOP           4             ;4 delay slots for LDH
              MPY           A2,B2,A3      ;A3 = x * y
              NOP                         ;1 delay slot for MPY
              ADD           A3,A7,A7      ;sum of products in A7
              SUB           A1,1,A1       ;decrement loop counter
     [A1]     B             loop          ;branch back to loop till A1=0
              NOP           5             ;5 delay slots for branch

              MV            A7,A4         ;A4=result A4=return register
              B             B3            ;return from func to addr in B3
              NOP           5             ;5 delay slots for branch
```

图 3.13 由汇编程序调用求点积的汇编函数（dotp4afunc.asm）

```
LDH *B4++,B2
```

把 B4（第二个数组的地址）所指定的存储器（第二个数组中的第一个数的地址是 y_address）中的内容赋给 B2，然后指针寄存器 B4 指向下一个高地址，即第二个数组中的第二个数。寄存器 A7 用于累加和传送积的和到寄存器 A4，因为最终结果要通过 A4 传给调用函数。

该工程的支持文件有（不需要库函数）：

1. dotp4a_init.asm
2. dotp4afunc.asm
3. vectors_dotp4a.asm

```
;vectors_dotp4a.asm Vector file for dotp4a project

        .ref      init       ;starting addr in init file
        .sect     "vectors"  ;in section vectors
rst:    mvkl  .s2 init,b0    ;init addr 16 LSB --->B0
        mvkh  .s2 init,b0    ;init addr 16 MSB --->B0
        b         b0         ;branch to addr init
        nop
        nop
        nop
        nop
        nop
```

图 3.14　求点积的矢量文件，在该文件中指定在调用汇编程序中的入口地址（vectors_dotp4a.asm）

建立并运行名为 dotp4a 的工程，修改连接选项（在 Project→Options 菜单里），选择“No Autoinitialization”，否则在建立工程时，会出现“entry point symbol _c_int00 undefined”警告（该警告可忽略），这是因为在该工程中，没有 C 程序主函数，因此没有通常程序运行的起始点。

在程序 dotp4a_init.asm 的第一个分支指令处设置断点：

```
B dotp4afunc
```

选择菜单 View→Memory，使用 16 位有符号整型数，设置地址为 result_addr。在 Memory 窗口向右点击，并取消选择“Float in Main Window”，这样，当查看源文件 dotp4a_init.asm 时，能更好地观察存储器窗口。

选择菜单 Run，在断点处程序停止执行，地址为 result_addr 的存储器中的内容为 0（被调用函数 dotp4afunc.asm 尚未执行）。再继续运行，然后停下来（因为程序执行在无限等待循环指令内）：

```
wait  B  wait  ;wait here
```

检验当前积的和是否为 40。注意：寄存器 A0 的内容是结果的地址（result_addr）。在菜单中选择 View→CPU Registers→Core Registers，检验该地址（十六进制格式）。图 3.15 给出该工程的 CCS 显示窗口，从反汇编文件中，可看出程序停止在无限等待循环语句中。

例 3.5　使用 C 函数调用线性汇编函数求点积

图 3.16 给出了 C 程序 dotp4clasm.c，它调用线性汇编函数 dotp4clasmfunc.sa（如图 3.17 所示）。例 1.3 介绍了只用 C 语言程序实现点积运算，前面三个例子介绍了汇编程序的语法。

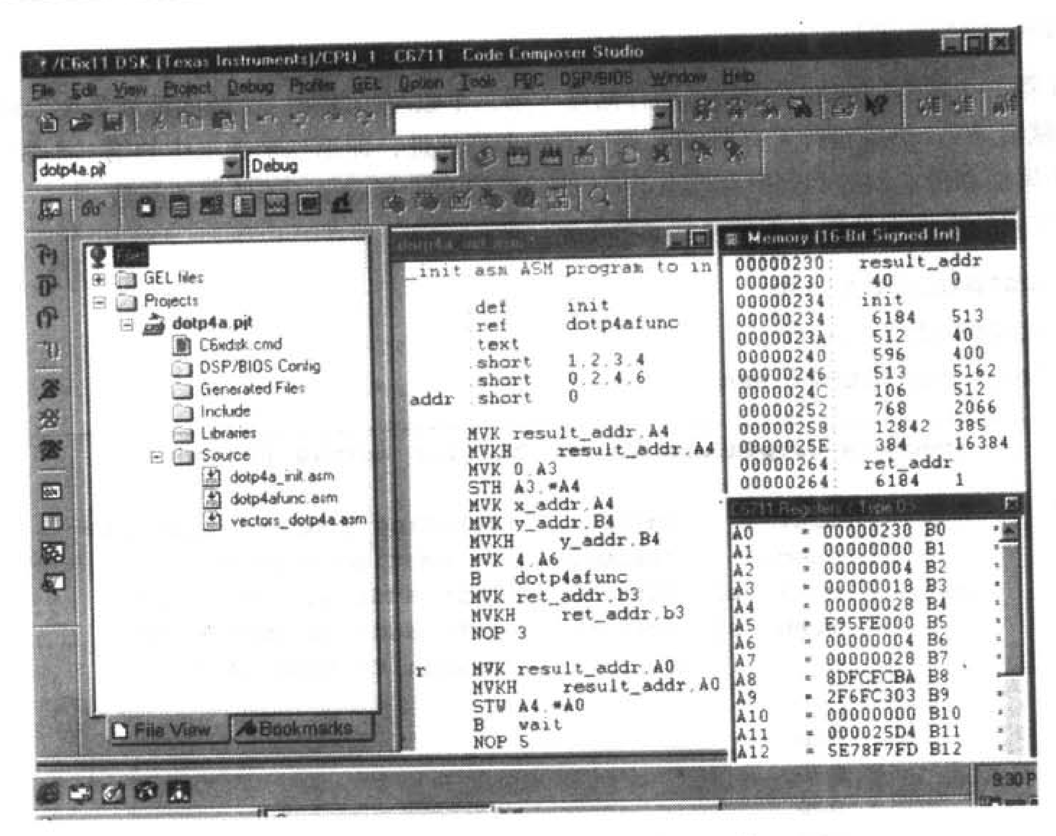


图 3.15　工程 dotp4a 中求点积的 CCS 窗口

线性汇编优化器优化的程序部分分别用线性汇编器命令.cproc 和.endproc 表示线性汇编开始和结束。因为调用函数在 C 程序中，被调用的线性汇编函数名前要用下划线，汇编命令.ref(.def)指定参考（定义）函数。

```
//Dotp4clasm.c Multiplies two arrays using C calling linear ASM func

short dotp4clasmfunc(short *a,short *b,short ncount);   //prototype
#include <stdio.h>                                      //for printing statement
#include "dotp4.h"                                      //arrays of data values
#define   count 4                                       //number of data values
short x[count] = {x_array};                             //declare 1st array
short y[count] = {y_array};                             //declare 2nd array
volatile int result = 0;                                //result

main()
{
  result = dotp4clasmfunc(x,y,count);                   //call linear ASM func
  printf("result = %d decimal \n", result);             //print result
}
```

图 3.16　调用线性汇编函数求点积的 C 程序（dotp4clasm.c）

在汇编程序中，功能单元是可选的，寄存器 a，b，prod 和 sum 由线性汇编命令.reg 定义，两个数组的地址 x，y 和数组的维数通过寄存器 ap，bp 和 count 传送给线性汇编函数。和在 C 程序中一样，寄存器 ap 和 bp 用做指针，语句的指令部分看上去像汇编程序代码，但后面部分和 C 语

言编程语法一样。例如，指令：

```
loop:    ldh    *ap++,a
```

（第一次通过程序的循环部分）把寄存器 ap 的内容作为存储器地址，并将该存储器的内容传送到寄存器 a 中，然后指针寄存器 ap 递增，指向下一个较高存储器地址，即指向数组 x 中第二个数的存储单元的地址，积的和在 sum 中累加，再返回给调用 C 程序。

```
;Dotp4clasmfunc.sa Linear assembly function to multiply two arrays
                   .ref     _dotp4clasmfunc ;ASM func called from C
_dotp4clasmfunc:   .cproc   ap,bp,count     ;start section linear asm
                   .reg     a,b,prod,sum    ;asm optimizer directive

                   zero     sum             ;init sum of products
loop:              ldh      *ap++,a         ;pointer to 1st array->a
                   ldh      *bp++,b         ;pointer to 2nd array->b
                   mpy      a,b,prod        ;product= a*b
                   add      prod,sum,sum    ;sum of products-->sum
                   sub      count,1,count   ;decrement counter
  [count]          b        loop            ;loop back if count # 0

                   .return  sum             ;return sum as result
                   .endproc                 ;end linear asm function
```

图 3.17 由 C 程序调用求点积的线性汇编函数（dotp4clasmfunc.sa）

建立并运行工程 dotp4clasm，检验下面的输出结果：result = 40。如果有兴趣的话，可以观察一下线性汇编函数，比较一下它和例 1.3 中 C 程序的执行时间差别。

例 3.6 用 C 程序调用线性汇编函数求阶乘

图 3.18 给出了 C 程序 factclasm.c，它调用了线性汇编函数 factclasmfunc.sa（如图 3.19 所示），用于计算小于 8 的数的阶乘。也可查看例 3.3，它利用 C 程序调用汇编函数来计算数的阶乘。例 3.5 利用 C 程序调用线性汇编函数来计算积的和，它对该例有参考意义。例 3.3 和例 3.5 包括了该例必要的背景知识。

该工程的支持文件有：factclasm.c，factclasmfunc.sa，vectors，rts6701.lib 和 C6xdsk.cmd。建立并运行工程 factclasm，检验 7! 的输出结果，也就是检查结果是否是 factorial = 5040。

```
//Factclasm.c Factorial of number. Calls linear ASM function

#include <stdio.h>                    //for print statement

void main()
{
 short number = 7;                    //set value
 short result;                        //result of factorial

 result = factclasmfunc(number);      //call ASM function factlasmfunc
 printf("factorial = %d", result);   //print from linear ASM function
}
```

图 3.18 调用线性汇编函数求阶乘的 C 程序（factclasm.c）

```
;Factclasmfunc.sa Linear ASM function called from C to find factorial

                  .ref      _factclasmfunc  ;Linear ASM func called from C
_factclasmfunc:   .cproc    number          ;start of linear ASM function
                  .reg      a,b             ;asm optimizer directive
                  mv        number,b        ;set-up loop count in b
                  mv        number,a        ;move number to a
                  sub       b,1,b           ;decrement loop counter

loop:             mpy       a,b,a           ;n(n-1)
                  sub       b,1,b           ;decrement loop counter
   [b]            b         loop            ;loop back to loop if count # 0
                  .return a                 ;result to calling function
                  .endproc                  ;end of linear asm function
```

图 3.19 被 C 程序调用求阶乘的线性汇编函数（factclasmfunc.sa）

参考文献

1. R. Chassaing and D. W. Horning, *Digital Signal Processing with the TMS320C25*, Wiley, New York, 1990.
2. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, Wiley, New York, 1999.
3. R. Chassaing, *Digital Signal Processing with C and the TMS320C30*, Wiley, New York, 1992.
4. R. Chassaing and P. Martin, Parallel processing with the TMS320C40, *Proceedings of the 1995 ASEE Annual Conference*, June 1995.
5. R. Chassaing and R. Ayers, Digital signal processing with the SHARC, *Proceedings of the 1996 ASEE Annual Conference*, June 1996.
6. *TMS320C6000 CPU and Instruction Set*, SPRU189F, Texas Instruments, Dallas, TX, 2000.
7. *TMS320C6000 Peripherals*, SPRU190D, Texas Instruments, Dallas, TX, 2001.
8. *TMS320C6000 Programmer's Guide*, SPRU198D, Texas Instruments, Dallas, TX, 2000.
9. *TMS320C6000 Assembly Language Tools User's Guide*, SPRU186G, Texas Instruments, Dallas, TX, 2000.
10. *TMS320C6000 Optimizing Compiler User's Guide*, SPRU187G, Texas Instruments, Dallas, TX, 2000.
11. *TMS320C6211 Fixed-Point Digital Signal Processor—TMS320C6711 Floating-Point Digital Signal Processor*, SPRS073C, Texas Instruments, Dallas, TX, 2000.

第 4 章　有限冲激响应滤波器

本章介绍了和离散时间信号联系在一起的 z 变换，阐述了由拉普拉斯变换的 s 平面映射到z变换的 z 平面，同时还介绍了用傅里叶级数方法，设计 FIR 滤波器以及编写离散卷积方程程序来实现 FIR 滤波器，最后讨论了窗函数对 FIR 滤波器特性的影响。本章需要掌握的内容包括 z 变换、有限冲激响应（FIR）滤波器的设计与实现及 C 语言和 TMS320C6x 汇编程序实例。

4.1　z 变换基础

与拉普拉斯变换用于连续时间信号分析相类似，z 变换用于分析离散时间信号。我们可以用拉普拉斯变换来求解表示模拟滤波器的微分方程，或者用 z 变换来求解表示数字滤波器的差分方程。考察一个模拟信号$x(t)$，当它被理想抽样后变为$x_s(t)$：

$$x_s(t)=\sum_{k=0}^{\infty}x(t)\delta(t-kT) \tag{4.1}$$

这里$\delta(t-kT)$是延迟为kT的冲激函数（δ函数），$T=1/F_s$为抽样周期。函数$x_s(t)$除了在$t=kT$时刻的值不为 0 外，其余时刻的值都为 0。$x_s(t)$的拉普拉斯变换为：

$$\begin{aligned}X_s(s)&=\int_0^{\infty}x_s(t)e^{-st}dt\\&=\int_0^{\infty}\{x(t)\delta(t)+x(t)\delta(t-T)+\cdots\}e^{-st}dt\end{aligned} \tag{4.2}$$

根据冲激函数的性质：

$$\int_0^{\infty}f(t)\delta(t-kT)dt=f(kT)$$

式（4.2）中的$X_s(s)$可写为：

$$X_s(s)=x(0)+x(T)e^{-sT}+x(2T)e^{-2sT}+\cdots=\sum_{n=0}^{\infty}x(nT)e^{-nsT} \tag{4.3}$$

令式（4.3）中$z=e^{sT}$，则式（4.3）可写为：

$$X(z)=\sum_{n=0}^{\infty}x(nT)z^{-n} \tag{4.4}$$

将抽样周期 T 略去不写，则$x(nT)$可写成$x(n)$，这样式（4.4）可写为：

$$X(z)=\sum_{n=0}^{\infty}x(n)z^{-n}=ZT\{x(n)\} \tag{4.5}$$

式（4.5）表示$x(n)$的 z 变换(ZT)，$x(n)$与$X(z)$是一一对应的，所以 z 变换是一种独特的变换。

练习 4.1　求指数函数$x(n)=e^{nk}$的 z 变换

若$n\geqslant 0$，且 k 为常数，则指数函数$x(n)=e^{nk}$的 z 变换为：

$$X(z)=\sum_{n=0}^{\infty}e^{nk}z^{-n}=\sum_{n=0}^{\infty}(e^{k}z^{-1})^{n} \tag{4.6}$$

由泰勒级数近似得到的几何级数：

$$\sum_{n=0}^{\infty}u^{n}=\frac{1}{1-u} \qquad |u|<1$$

式（4.6）变为：

$$X(z)=\frac{1}{1-e^{k}z^{-1}}=\frac{z}{z-e^{k}} \tag{4.7}$$

其中$|e^{k}z^{-1}|<1$或$|z|>|e^{k}|$。$k=0$时，$x(n)=1$的z变换是$X(z)=z/(z-1)$。

练习 4.2　求正弦序列$x(n)=\sin n\omega T$的z变换

正弦函数可用复指数形式表示，由欧拉公式$e^{ju}=\cos u+j\sin u$，可得：

$$\sin n\omega T=\frac{e^{jn\omega T}-e^{-jn\omega T}}{2j}$$

则：

$$X(z)=\frac{1}{2j}\sum_{n=0}^{\infty}(e^{jn\omega T}z^{-n}-e^{-jn\omega T}z^{-n}) \tag{4.8}$$

利用练习 4.1 中的几何级数，可以求出$X(z)$。或直接利用式（4.7）的结果，将式（4.8）的第一项求和式中用$k=j\omega T$，第二项求和式中用$k=-j\omega T$代入，可得：

$$\begin{aligned}X(z)&=\frac{1}{2j}\left(\frac{z}{z-e^{j\omega T}}-\frac{z}{z-e^{-j\omega T}}\right)\\&=\frac{1}{2j}\frac{z^{2}-ze^{-j\omega T}-z^{2}+ze^{j\omega T}}{z^{2}-z(e^{-j\omega T}+e^{j\omega T})+1}\\&=\frac{z\sin\omega T}{z^{2}-2z\cos\omega T+1}\end{aligned} \tag{4.9}$$

$$=\frac{Cz}{z^{2}-Az-B} \qquad |z|>1 \tag{4.10}$$

这里$A=2\cos\omega T, B=-1, C=\sin\omega T$。在第 5 章中，我们基于该结果产生了一个正弦信号，通过改变式（4.9）中的ω值，我们可以很容易产生不同频率的正弦波形。

同样道理，利用欧拉公式将$\cos n\omega T$表示成两个复指数的和，就可以得出$x(n)=\cos n\omega T=(e^{jn\omega T}+e^{-jn\omega T})/2$的$z$变换，即：

$$X(z)=\frac{z^{2}-z\cos\omega T}{z^{2}-2z\cos\omega T+1} \qquad |z|>1 \tag{4.11}$$

4.1.1　s 平面到 z 平面的映射

拉普拉斯变换可以用来判断一个系统的稳定性。如果一个系统的极点在 s 平面上都位于虚轴的左侧，该系统响应是随时间衰减的稳定的系统；如果极点位于虚轴的右侧，其系统响应随时间增大，所以该系统是不稳定的；如果极点位于虚轴上，即极点为纯虚数，则其响应为正弦响应，且频率由$j\omega$轴表示，$\omega=0$表示为直流。

同样，我们可以根据系统经过 z 变换后，其极点在 z 平面上的位置来判断该系统的稳定性，因为我们可以找到 s 平面与 z 平面相对应区域。由于 $z=e^{sT}$ 以及 $s=\sigma+j\omega$，则有：

$$z=e^{\sigma T}e^{j\omega T} \tag{4.12}$$

因此，z 的幅度为 $|z|=e^{\sigma T}$，相位为 $\theta=\omega T=2\pi f/F_s$，其中 F_s 为抽样频率。为了说明 s 平面到 z 平面的映射，参见图 4.1 所示区域的对应关系。

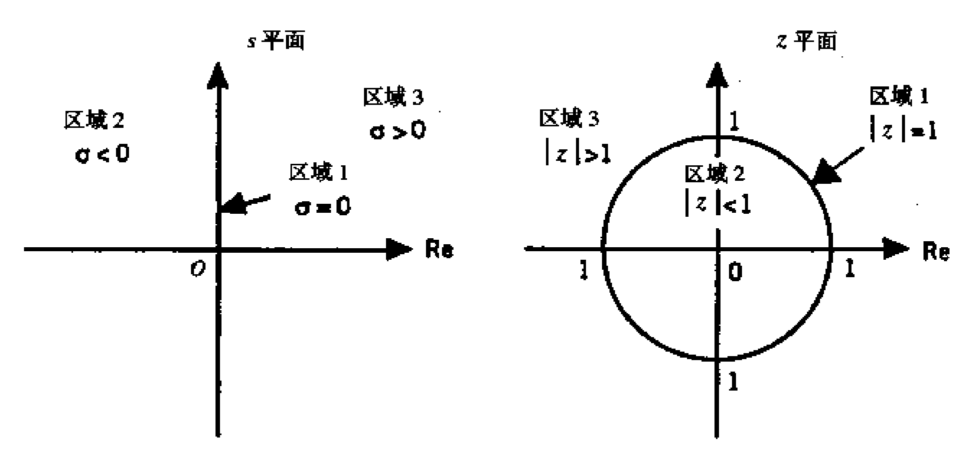


图 4.1　自 s 平面至 z 平面的映射

σ<0

极点位于 s 平面虚轴的左侧（即图 4.1 中区域 2）的系统是稳定的系统，因为 $e^{\sigma T}<1$，式（4.12）产生的幅度 $|z|<1$。随着 σ 从 $-\infty$ 变到 0^-，$|z|$ 从 0 变到 1^-，因此，极点在 z 平面区域 2 单位圆内的系统是稳定系统。对于这样的系统，如果极点是实数，则其响应为指数衰减的；如果极点为复数，则响应为正弦衰减信号。

σ>0

极点位于 s 平面虚轴右侧（即图 4.1 中区域 3）的系统是不稳定系统，因为 $e^{\sigma T}>1$，式（4.12）产生的幅度 $|z|>1$。随着 σ 从 0^+ 变到 ∞，$|z|$ 从 1^+ 变到 ∞，因此，极点在 z 平面区域 3 单位圆外的系统是不稳定系统。对于这样的系统，如果极点是实数，则其响应为指数增加的；如果极点为复数，则响应为幅度逐步增大的正弦信号。

σ=0

极点位于 s 平面虚轴上（即图 4.1 中区域 1）的系统是临界稳定系统，式（4.12）产生的幅度 $|z|=1$，对应区域 1（单位圆），因此，极点在 z 平面的单位圆上会使系统产生正弦振荡。第 5 章中，我们通过编程使差分方程的极点在单位圆上来实现正弦信号。注意：练习 4.2 中，式（4.9）中 $X(s)=\sin n\omega T$ 的极点或式（4.11）中 $X(s)=\cos n\omega T$ 的极点是 $z^2-2z\cos\omega T+1$ 的根，即：

$$\begin{aligned}p_{1,2}&=\frac{2\cos\omega T\pm\sqrt{4\cos^2\omega T-4}}{2}\\&=\cos\omega T\pm\sqrt{-\sin^2\omega T}=\cos\omega T\pm j\sin\omega T\end{aligned} \tag{4.13}$$

每个极点的模为：

$$|p_1|=|p_2|=\sqrt{\cos^2\omega T+\sin^2\omega T}=1 \tag{4.14}$$

z 的相位为 $\theta=\omega T=2\pi f/F_s$，当频率 f 从 0 变到 $\pm F_s/2$，相位 θ 从 0 变到 π。

4.1.2 差分方程

就像模拟滤波器可以用微分方程来表示一样，数字滤波器可用差分方程来表示。为了求解差分方程，我们要找到形如$x(n-k)$表达式的z变换，$x(n-k)$对应于模拟信号$x(t)$的 k 阶导数$d^k x(t)/dt^k$，差分方程的阶数由k的最大值决定，例如$k=2$表示二阶差分方程。由式（4.5）可得：

$$X(z)=\sum_{n=0}^{\infty}x(n)z^{-n}=x(0)+x(1)z^{-1}+x(2)z^{-2}+\cdots \tag{4.15}$$

对应于一阶导数dx/dt，可得$x(n-1)$的z变换为：

$$\begin{aligned}ZT[x(n-1)]&=\sum_{n=0}^{\infty}x(n-1)z^{-n}\\&=x(-1)+x(0)z^{-1}+x(1)z^{-2}+x(2)z^{-3}+\cdots\\&=x(-1)+z^{-1}[x(0)+x(1)z^{-1}+x(2)z^{-2}+\cdots]\\&=x(-1)+z^{-1}X(z)\end{aligned} \tag{4.16}$$

上式中使用式（4.15），$x(-1)$表示一阶差分方程的初始条件。类似地，相应于二阶导数$d^2x(t)/dt^2$，$x(n-2)$的z变换为：

$$\begin{aligned}ZT[x(n-2)]&=\sum_{n=0}^{\infty}x(n-2)z^{-n}\\&=x(-2)+x(-1)z^{-1}+x(0)z^{-2}+x(1)z^{-3}+\cdots\\&=x(-2)+x(-1)z^{-1}+z^{-2}[x(0)+x(1)z^{-1}+\cdots]\\&=x(-2)+x(-1)z^{-1}+z^{-2}X(z)\end{aligned} \tag{4.17}$$

$x(-2)$与$x(-1)$表示求解二阶差分方程所需的两个初始条件。一般地：

$$ZT[x(n-k)]=z^{-k}\sum_{m=1}^{k}x(-m)z^{m}+z^{k}X(z) \tag{4.18}$$

如果初始条件都为 0，即$x(-m)=0$，$m=1,2,\cdots,k$，这时式（4.18）可简化为：

$$ZT[x(n-k)]=z^{-k}X(z) \tag{4.19}$$

4.2 离散信号

离散信号可表示为：

$$x(n)=\sum_{m=-\infty}^{\infty}x(m)\delta(n-m) \tag{4.20}$$

这里$\delta(n-m)$表示延迟为m的冲激序列，当$n=m$时，它等于 1，否则等于 0。它由值为$x(1),x(2),\cdots$的序列组成，n 表示时间，序列中的每个抽样值之间相差一个抽样时间间隔，该时间由抽样间隔或抽样周期$T=1/F_s$决定。

本书中所讨论的信号与系统都是线性时不变的，即叠加和时移不变特性都适用。设输入信号为$x(n)$，输出响应为$y(n)$，即$x(n)\rightarrow y(n)$。如果$a_1x_1(n)\rightarrow a_1y_1(n)$，$a_2x_2(n)\rightarrow a_2y_2(n)$，则$a_1x_1(n)+a_2x_2(n)\rightarrow a_1y_1(n)+a_2y_2(n)$，这里$a_1$，$a_2$是常数，这就是叠加特性，即总输出响应为单个响应之和。时移不变性是指如果输入延迟m个抽样时间，输出响应同样会延迟m个抽样时间。即

$x(n-m) \to y(n-m)$。如果输入为单位冲激$\delta(n)$，那么输出响应就是$h(n)$，也就是$\delta(n) \to h(n)$，$h(n)$被认为是系统的冲激响应。根据时移不变特性，延迟的冲激输入$\delta(n-m)$产生延迟的输出响应$h(n-m)$。

更进一步，如果该冲激与$x(m)$相乘，则$x(m)\delta(n-m) \to x(m)h(n-m)$，根据式（4.20），该响应变为：

$$y(n)=\sum_{m=-\infty}^{\infty} x(m)h(n-m) \tag{4.21}$$

即$y(n)$为$x(n)$与$h(n)$的卷积。对于一个因果系统，式（4.21）变为：

$$y(n)=\sum_{m=-\infty}^{\infty} x(m)h(n-m) \tag{4.22}$$

令式（4.22）中$k=n-m$，上式可写为：

$$y(n)=\sum_{k=0}^{\infty} h(k)x(n-k) \tag{4.23}$$

4.3　有限冲激响应滤波器

滤波是最常用的信号处理方式之一[1-4]，现在可用数字信号处理器实现实时数字滤波处理，TMS320C6x 的指令系统和结构使它们非常适合这种滤波运算。模拟滤波器对连续信号进行处理，一般利用诸如放大器、电阻和电容等分立元器件来实现；而数字滤波器（如 FIR 滤波器）对离散时间信号进行处理，可用如 TMS320C6x 这样的数字信号处理芯片来实现。这种过程涉及到用 ADC 采集外部输入信号，处理输入抽样值，然后将最后的结果通过 DAC 输出。

最近几年里，数字信号处理器的成本显著降低，这样使数字滤波器比对应的模拟滤波器有了更多的优点，其中包括可靠性高、精度高、对温度和老化不敏感等。使用数字滤波器可实现严格的幅度和相位特性。滤波器的各种特性，如中心频率、带宽和滤波器类型可以方便地调整。利用 TMS320C6x 的 DSK 和许多设计和实现工具，在几分钟内就可以实现实时 FIR 滤波器。滤波器的设计就是利用一组系数去逼近系统的传输函数。

可用多种不同的技术来设计 FIR 滤波器，如在 4.4 节中讨论的利用傅里叶级数的方法，它是一种常用的方法。计算机辅助设计技术，如 Parks 和 McClellan 的设计技术，也可用于 FIR 滤波器的设计[5, 6]。

式（4.23）的卷积方程对设计 FIR 滤波器非常有用，因为我们可以用有限项去逼近它，即：

$$y(n)=\sum_{k=0}^{N-1} h(k)x(n-k) \tag{4.24}$$

如果输入为单位冲激$x(n)=\delta(0)$，输出冲激响应就是$y(n)=h(n)$。在 4.4 节中，我们将看到如何利用N个系数$h(0), h(1), \cdots, h(N-1)$和$N$个输入抽样值$x(0), x(1), \cdots, x(n-(N-1))$来设计FIR 滤波器。$n$ 时刻的输入抽样值为$x(n)$，延时的输入抽样分别为$x(n-1), \cdots, x(n-(N-1))$。由式（4.24）可看出，一个 FIR 滤波器可根据 n 时刻的输入$x(n)$和多个延时的$x(n-k)$的信息来实现。它是非递归的，不需要反馈和过去的输出。需要过去输出的有反馈（递归）的滤波器将在第 5 章讨论。FIR 滤波器有时也被称为横向抽头延迟滤波器。

在初始条件为 0 情况下，式（4.24）的z变换为：

$$Y(z)=h(0)X(z)+h(1)z^{-1}X(z)+h(2)z^{-2}X(z)+\cdots+h(N-1)z^{-(N-1)}X(z) \quad (4.25)$$

式（4.24）表示系数与输入在时域上的卷积，它等效于在频域内相乘，即：

$$Y(z)=H(z)X(z) \quad (4.26)$$

其中$H(z)=ZT[h(k)]$是传递函数，即：

$$\begin{aligned}H(z)&=\sum_{k=0}^{N-1}h(k)z^{-k}=h(0)+h(1)z^{-1}+h(2)z^{-2}+\cdots+h(N-1)z^{-(N-1)}\\&=\frac{h(0)z^{(N-1)}+h(1)z^{N-2}+h(2)z^{N-3}+\cdots+h(N-1)}{z^{N-1}}\end{aligned} \quad (4.27)$$

式（4.27）表示$H(z)$有$N-1$个极点，所有极点都位于原点，因此，FIR滤波器是固有稳定的，它所有的极点都位于单位圆内，因此有时把FIR滤波器描述为“无极点”滤波器。图4.2给出了式（4.24）和式（4.25）代表的FIR滤波器结构。

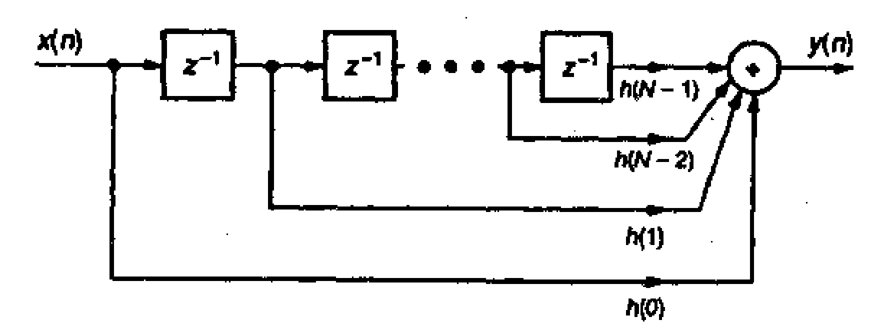


图4.2 显示延迟的FIR滤波器结构

FIR滤波器的一个非常有用的特性就是线性相位特性。线性相位特性在语音分析方面很有用，语音分析里对相位失真要求非常苛刻。例如，在线性相位情况下，所有输入正弦分量都有相同的延迟，否则，就会产生谐波失真。

延迟输入抽样$x(n-k)$的傅里叶变换是$e^{-j\omega kT}X(j\omega)$，产生的相移$\theta=-\omega kT$，它是频率$\omega$的线性函数。相位的导数定义为群延时，注意它是一个常数，即$d\theta/d\omega=-kT$。

4.4 利用傅里叶级数实现FIR滤波器

利用傅里叶级数法设计FIR滤波器是用传输函数$H(z)$的幅度响应去逼近要求的幅度响应。要求的传输函数为：

$$H_d(\omega)=\sum_{n=-\infty}^{\infty}C_n e^{jn\omega T} \qquad |n|<\infty \quad (4.28)$$

其中，C_n为傅里叶级数的系数，利用归一化频率变量v，$v=f/F_N$，这里F_N为奈奎斯特频率，即$F_N=F_s/2$，则式（4.28）所要求的传输函数可写为：

$$H_d(v)=\sum_{n=-\infty}^{\infty}C_n e^{jn\pi v} \quad (4.29)$$

其中$\omega T=2\pi f/F_s=\pi v$，$|v|<1$，系数$C_n$定义为：

$$C_n = \frac{1}{2}\int_{-1}^{1} H_d(v)e^{-jn\pi v}dv$$
$$= \frac{1}{2}\int_{-1}^{1} H_d(v)(\cos n\pi v - j\sin n\pi v)dv \tag{4.30}$$

假设$H_d(v)$为偶函数（频率选择性滤波器），则式（4.30）可简化为：

$$C_n = \int_0^1 H_d(v)\cos n\pi v dv \qquad n\geqslant 0 \tag{4.31}$$

因为$H_d(v)\ \sin n\pi v$是奇函数，且$C_n = C_{-n}$，则有：

$$\int_{-1}^{1} H_d(v)\sin n\pi v dv = 0$$

式（4.29）中，要求的传输函数$H_d(v)$用无限项系数来表示。为了获得可实现的滤波器，我们必须截短式（4.29），这样就形成了逼近的传输函数：

$$H_a(v) = \sum_{n=-Q}^{Q} C_n e^{jn\pi v} \tag{4.32}$$

其中 Q 是有限的正数，并且决定滤波器的阶数。Q 越大，FIR 滤波器的阶数越高，式（4.32）就越能逼近要求的传输函数。用有限项来截短无穷级数，忽略$-Q$ 到$+Q$ 矩形窗函数之外各项的作用。在 4.5 节中，我们会了解到采用矩形窗以外的窗函数可改进滤波器特性。

令$z = e^{j\pi v}$，这样式（4.32）变为：

$$H_a(z) = \sum_{n=-Q}^{Q} C_n z^n \tag{4.33}$$

冲激响应系数为C_{-Q}，C_{-Q+1}，…，C_{-1}，C_0，C_1，…，C_{Q-1}，C_Q。式（4.33）近似的传输函数中，z 的指数有正数，这意味着滤波器在施加输入信号前就有输出，因此滤波器是非因果、不可实现的。为了弥补这种情况，我们在式（4.33）中引入了 Q 个抽样的延迟，这样就形成了：

$$H(z) = z^{-Q}H_a(z) = \sum_{n=-Q}^{Q} C_n z^{n-Q} \tag{4.34}$$

令$n - Q = -i$，式（4.34）中的$H(z)$变为：

$$H(z) = \sum_{i=0}^{2Q} C_{Q-i} z^{-i} \tag{4.35}$$

令$h_i = C_{Q-i}$，$N - 1 = 2Q$，这样式 $H(z)$变为：

$$H(z) = \sum_{i=0}^{N-1} h_i z^{-i} \tag{4.36}$$

这里 $H(z)$用冲激响应系数 h_i 来表示，且$h_0 = C_Q, h_1 = C_{Q-1},\cdots,h_Q = C_0, h_{Q+1} = C_{-1} = C_1,\cdots,h_{2Q} = C_{-Q}$，冲激响应系数是关于$h_Q$ 对称的，且$C_n = C_{-n}$。

滤波器的阶数是$N = 2Q + 1$，例如，如果$Q = 5$，滤波器就有 11 个系数$h_0, h_1,\cdots, h_{10}$，即：

$$h_0 = h_{10} = C_5$$
$$h_1 = h_9 = C_4$$
$$h_2 = h_8 = C_3$$
$$h_3 = h_7 = C_2$$

$$h_4 = h_6 = C_1$$
$$h_5 = C_0$$

图 4.3 给出了频率选择性滤波器的理想传输函数，包括低通、高通、带通和带阻，其中可以发现系数$C_n = C_{-n}$。

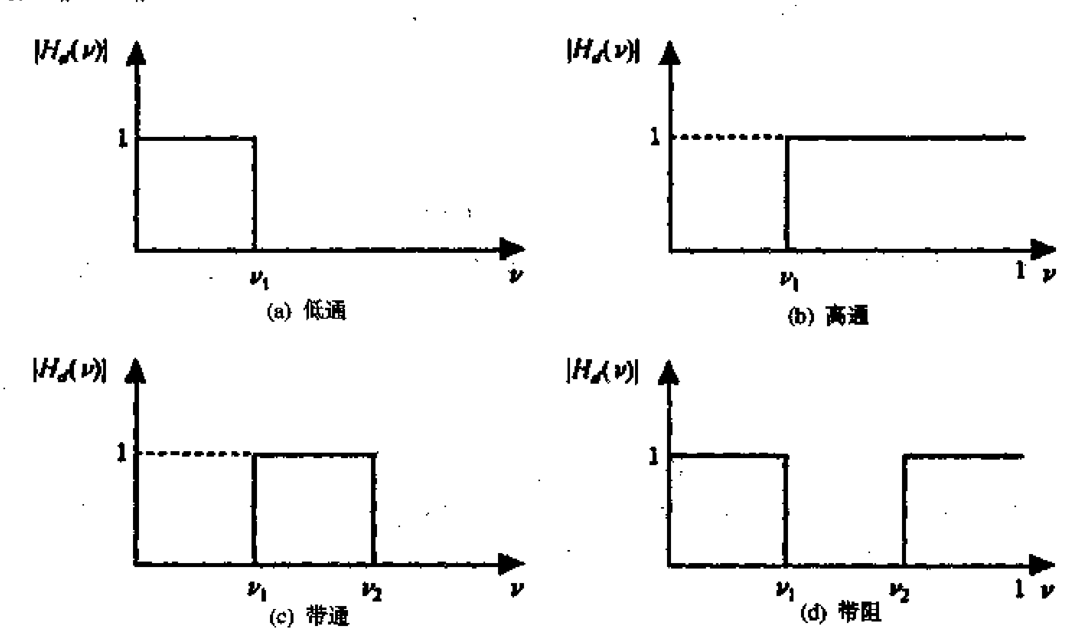


图 4.3 理想传输函数

1. 低通：$C_0 = \nu_1$

$$C_n = \int_0^{\nu_1} H_d(\nu)\cos n\pi\nu \, d\nu = \frac{\sin n\pi\nu_1}{n\pi} \tag{4.37}$$

2. 高通：$C_0 = 1 - \nu_1$

$$C_n = \sum_{\nu_1}^{1} H_d(\nu)\cos n\pi\nu \, d\nu = -\frac{\sin n\pi\nu_1}{n\pi} \tag{4.38}$$

3. 带通：$C_0 = \nu_2 - \nu_1$

$$C_n = \int_{\nu_1}^{\nu_2} H_d(\nu)\cos n\pi\nu \, d\nu = \frac{\sin n\pi\nu_2 - \sin n\pi\nu_1}{n\pi} \tag{4.39}$$

4. 带阻：$C_0 = 1 - (\nu_2 - \nu_1)$

$$C_n = \int_0^{\nu_1} H_d(\nu)\cos n\pi\nu \, d\nu + \int_{\nu_2}^{1} H_d(\nu)\cos n\pi\nu \, d\nu = \frac{\sin n\pi\nu_1 - \sin n\pi\nu_2}{n\pi} \tag{4.40}$$

其中图 4.3 中所示的ν_1和ν_2是归一化的截止频率。目前有几个工具包可用于 FIR 滤波器的设计，后面将要讨论它们。在实现 FIR 滤波器时，我们开发了一个通用程序，滤波器的类型（例如，无论低通还是带通）由特定的系数确定。

练习 4.3 低通 FIR 滤波器

求一个 FIR 滤波器的冲激响应的系数，其中阶数$N = 11$，抽样频率为 10 kHz，截止频率为$f_c = 1\,\text{kHz}$。由式（4.37）：

$$C_0 = \nu_1 = \frac{f_c}{F_N} = 0.2$$

这里$F_N = F_s/2$是奈奎斯特频率，且：

$$C_n = \frac{\sin 0.2n\pi}{n\pi} \qquad n = \pm 1, \ \pm 2, \ \cdots, \ \pm 5 \tag{4.41}$$

因为冲激响应系数$h_i = C_{Q-i}$和$C_n = C_{-n}$，因此冲激响应的系数为：

$$\begin{aligned} &h_0 = h_{10} = 0 && h_3 = h_7 = 0.1514 \\ &h_1 = h_9 = 0.0468 && h_4 = h_6 = 0.1872 \\ &h_2 = h_8 = 0.1009 && h_5 = 0.2 \end{aligned} \tag{4.42}$$

这些系数可利用实用程序（见辅助材料）计算出来，再插入到一个通用滤波器程序里，后面将讨论这些工具软件。注意这些系数是关于$Q = 5$对称的。对于一个实际的滤波器，阶数$N = 11$是较低的，阶数N加倍就可以得到有更好特性的 FIR 滤波器，比如在选择性方面。对于有线性相位特性的 FIR 滤波器，如式（4.42）所示，系数必须是对称的。

4.5　窗函数

对式（4.29）传输函数的无穷级数进行截短，得到式（4.32）截短传输函数。该过程实质上是在$-Q$到$+Q$之间加了一个幅度为 1 的矩形窗函数，并忽略了窗外的各个系数。矩形窗越宽，Q值越大，式（4.32）中所取的项数也越多，也就能更好地逼近式（4.29），因此矩形窗函数可定义为：

$$w_R(n) = \begin{cases} 1 & |n| \leqslant Q \\ 0 & \text{其他} \end{cases} \tag{4.43}$$

矩形窗函数$\omega_R(n)$变换到频域是 sinc 函数，它在频域可表示为：

$$W_R(\nu) = \sum_{n=-Q}^{Q} e^{jn\pi\nu} = e^{-jQ\pi\nu}\left(\sum_{n=0}^{2Q} e^{jn\pi\nu}\right) = \frac{\sin\left[\left(\frac{2Q+1}{2}\right)\pi\nu\right]}{\sin(\pi\nu/2)} \tag{4.44}$$

由于突然截断，特别是接近不连续，造成了 sinc 函数的旁瓣高或振荡的特点。

目前有多种窗函数被用来减小大幅度的振荡，它们在对无限长序列展开式截短时，采用了更加平滑的截短方式。然而，尽管其他的窗函数减小旁瓣的幅度，但同时也导致主瓣变宽，从而使滤波器选择性降低。衡量滤波器性能的一个参数是波动因子，它是将第一旁瓣峰值与主瓣峰值的进行比较（即它们的比率）。折衷的方法是选择一种窗函数，它既能达到矩形窗函数的选择性，又能减小旁瓣。减小主瓣宽度可以通过增加窗函数的宽度（增加滤波器阶数）来实现。后面我们将画出一个 FIR 滤波器的幅频响应及其不理想的旁瓣特性。

总之，傅里叶级数的系数可写为：

$$C'_n = C_n w(n) \tag{4.45}$$

其中$w(n)$为窗函数。在矩形窗函数情况下，$C'_n = C_n$，式（4.36）中的传输函数可写为：

$$H'(z) = \sum_{i=0}^{N-1} h'_i z^{-i} \tag{4.46}$$

其中：

$$h'_i = C'_{Q-i} \qquad 0 \leqslant i \leqslant 2Q \tag{4.47}$$

矩形窗有最高的旁瓣电平，其第一旁瓣电平只比主瓣峰值低 13 dB，会导致较大幅度的振荡。另一方面，它有最窄的主瓣，可提供高选择性。下面几种窗函数是在 FIR 滤波器设计中经常用到的窗函数。

4.5.1 汉明（Hamming）窗

汉明窗函数[12,25]为：

$$w_H(n) = \begin{cases} 0.54 + 0.46\cos(n\pi/Q) & |n| \leqslant Q \\ 0 & \text{其他} \end{cases} \tag{4.48}$$

其最高的旁瓣电平或第一旁瓣电平约比主瓣峰值低 43 dB。

4.5.2 汉宁（Hanning）窗

汉宁窗也称升余弦窗，其窗函数为：

$$w_{HA}(n) = \begin{cases} 0.5 + 0.5\cos(n\pi/Q) & |n| \leqslant Q \\ 0 & \text{其他} \end{cases} \tag{4.49}$$

其最高旁瓣电平或第一旁瓣电平约比主瓣峰值低 31 dB.

4.5.3 布莱克曼（Blackman）窗

布莱克曼窗函数是：

$$w_B(n) = \begin{cases} 0.42 + 0.5\cos(n\pi/Q) + 0.08\cos(2n\pi/Q) & |n| \leqslant Q \\ 0 & \text{其他} \end{cases} \tag{4.50}$$

其最高旁瓣电平约比主瓣峰值低 58 dB。与前几个窗函数相比，尽管布莱克曼窗对旁瓣有最大的衰减，但同时它的主瓣宽度也最宽。与上述的窗函数相同，主瓣宽度可通过增加窗函数的宽度来减小。

4.5.4 凯塞（Kaiser）窗

最近几年，用凯塞窗来设计 FIR 滤波器变得非常普遍。它有一个可变参数，可以用来控制旁瓣与主瓣相对大小。凯塞窗函数如下：

$$w_K(n) = \begin{cases} I_0(b)/I_0(a) & |n| \leqslant Q \\ 0 & \text{其他} \end{cases} \tag{4.51}$$

其中a是根据经验确定的变量，$b = a[1 - (n/Q)^2]^{1/2}$。$I_0(x)$是第一类修正贝塞尔函数，定义为：

$$I_0(x) = 1 + \frac{0.25x^2}{(1!)^2} + \frac{(0.25x^2)^2}{(2!)^2} + \cdots = 1 + \sum_{n=1}^{\infty}\left[\frac{(x/2)^n}{n!}\right]^2 \tag{4.52}$$

该函数能快速收敛。改变窗的宽度和参数a，可实现旁瓣电平与主瓣宽度的折中。

4.5.5 计算机辅助逼近设计

基于瑞梅兹（Remez）交换算法的计算机辅助迭代设计是一种有效的方法，利用该方法可设计出近似等波动特性的 FIR 滤波器[5,6]。滤波器的阶数及通带与阻带边界是固定的，通过改变系数来实现近似等波动特性，这使通带和阻带的波动都达到最小。对过渡带不加限制，并认为是“不用关心”的区域，没有办法解决这种情况。很多商业滤波器设计软件包使用 Parks-McClellan 算法来设计 FIR 滤波器。

4.6 C 语言和汇编程序编程实例

在几分钟内就可设计和实现一个实时的 FIR 滤波器，有很多滤波器设计软件包可用来设计 FIR 滤波器。附录 D 中介绍使用 MATLAB[48]来设计滤波器，附录 E 介绍使用 DigiFilter 软件和自己设计软件包（在辅助材料中）来设计滤波器。

有多个实例用来说明 FIR 滤波器的实现，多数是用 C 语言编写的，少数是用 C 语言与汇编混合编程，这些实例用来说明环形缓冲区的使用，利用内部或外部存储器中的环形缓冲区更新延时抽样点是一种更有效的方法。式（4.24）的卷积等式用于设计和实现滤波器，即：

$$y(n)=\sum_{i=0}^{N-1}h(i)x(n-i)$$

我们可以将冲激响应系数安排在一个缓冲区（数组）中，第一个系数$h(0)$位于缓冲区开始位置单元（存储器低端地址的第一个存储单元），最后一个系数$h(N-1)$位于缓冲区的最后位置单元（存储器高端地址的最后一个存储单元）。延时抽样在存储器中的安排是这样的：最新的抽样 $x(n)$ 位于抽样缓冲区的开始，最早的抽样$x(n-(N-1))$位于缓冲区的最后，系数和抽样在存储器中的组织如表 4.1 所示。开始时，所有抽样都设为 0。

表 4.1　系数和抽样初始时在存储器中的组织

i	系　数	示　例
0	$h(0)$	$x(n)$
1	$h(1)$	$x(n-1)$
2	$h(2)$	$x(n-2)$
.	.	.
.	.	.
.	.	.
$N-1$	$h(N-1)$	$x(n-(N-1))$

***n* 时刻**

在 n 时刻由 ADC 获得的最新抽样$x(n)$，存放在抽样缓冲区的开始位置单元，滤波器在 n 时刻的输出由卷积式（4.24）计算出来，即：

$$\begin{aligned}y(n)=&h(0)x(n)+h(1)x(n-1)+\cdots+h(N-2)x(n-(N-2))\\&+h(N-1)x(n-(N-1))\end{aligned}$$

延时抽样接着被不断更新，这样$x(n-k)=x(n+1-k)$就被用来计算$y(n+1)$，$y(n+1)$就是下一时间单元或者说是下一个抽样周期T_s的输出。除了最新的抽样外，所有抽样都要更新。例如，$x(n-1)=x(n)$，$x(n-(N-1))=x(n-(N-2))$，这种更新过程好像在存储器中把数据（向下）

移动似的（参见表4.2，观察$n+1$时刻数据的更新情况）。

$n+1$时刻

在$n+1$时刻获得的新输入抽样$x(n+1)$，将它存在抽样缓冲区的开始，如表4.2所示。输出$y(n+1)$可按下式计算：

$$y(n+1)=h(0)x(n+1)+h(1)x(n)+\cdots+h(N-2)x(n-(N-3))$$
$$+h(N-1)x(n-(N-2))$$

随后抽样更新为下一时间单元的抽样。

表4.2 更新的抽样在存储器中的组织

	示例		
系数	n时刻	$n+1$时刻	$n+2$时刻
$h(0)$	$x(n)$	$x(n+1)$	$x(n+2)$
$h(1)$	$x(n-1)$	$x(n)$	$x(n+1)$
$h(2)$	$x(n-2)$	$x(n-1)$	$x(n)$
.	.	.	.
.	.	.	.
.	.	.	.
$h(N-3)$	$x(n-(N-3))$	$x(n-(N-4))$	$x(n-(N-5))$
$h(N-2)$	$x(n-(N-2))$	$x(n-(N-3))$	$x(n-(N-4))$
$h(N-1)$	$x(n-(N-1))$	$x(n-(N-2))$	$x(n-(N-3))$

$n+2$时刻

在$n+2$时刻，得到新的输入抽样$x(n+2)$，输出变为：

$$y(n+2)=h(0)x(n+2)+h(1)x(n+1)+\cdots+h(N-1)x(n-(N-3))$$

在每个时间单元（抽样周期），更新延时抽样，计算滤波器的输出，该过程一直这样继续。例4.8说明了把系数和抽样存放在存储器中，并计算卷积等式的4种不同方法（例如，最新的抽样位于缓冲区的最后，而最早的抽样位于缓冲区的开始位置单元）。

例4.1 带阻和带通FIR滤波器的实现

图4.4给出了实现FIR滤波器的源程序（FIR.c）清单，这是一个通用的FIR程序，其中系数文件bs2700.cof（如图4.5所示）确定了滤波器的特性。系数文件有89个系数，表示中心频率为2700 Hz的FIR带阻滤波器，系数的个数N也定义在该文件中。滤波器设计利用MATLAB图形用户接口（GUI）的滤波器设计工具SPTOOL来设计的，附录D对该工具进行了介绍。图4.6给出了滤波器的特性（MATLAB的88阶滤波器对应89个系数）。

延时的抽样存放在缓冲区dly[N]里，最新的输入抽样$x(n)$由dly[0]得到并存放在缓冲区的开始处，而系数存放在另一个缓冲区h[N]中，h[0]在缓冲区的开始。表4.1给出了抽样和系数分别位于各自缓冲区中的组织方式。

在中断服务子程序中有两个for循环（将用一个循环实现FIR滤波器）。第一个循环实现在特定的n时刻N个系数和N个抽样的卷积。n时刻的输出为：

$$y(n)=h(0)x(n)+h(1)x(n-1)+\cdots+h(N-1)x(n-(N-1))$$

```
//Fir.c FIR filter. Include coefficient file with length N

#include "bs2700.cof"              //coefficient file BS @ 2700Hz
int yn = 0;                        //initialize filter's output
short dly[N];                      //delay samples

interrupt void c_int11()           //ISR
{
      short i;

      dly[0] = input_sample();     //newest input @ top of buffer
      yn = 0;                      //initialize filter's output
      for (i = 0; i< N; i++)
         yn += (h[i] * dly[i]);    //y(n) += h(i)* x(n-i)
      for (i = N-1; i > 0; i--)    //starting @ bottom of buffer
         dly[i] = dly[i-1];        //update delays with data move

      output_sample(yn >> 15);     //output filter
      return;
}

void main()
   {
      comm_intr();                 //init DSK, codec, McBSP
      while(1);                    //infinite loop
   }
```

图 4.4 通用 FIR 设计程序（FIR.c）

```
//BS2700.cof FIR bandstop coefficients designed with MATLAB

#define N 89                       //number of coefficients

short h[N]={-14,23,-9,-6,0,8,16,-58,50,44,-147,119,67,-245,
            200,72,-312,257,53,-299,239,20,-165,88,0,105,
           -236,33,490,-740,158,932,-1380,392,1348,-2070,
            724,1650,-2690,1104,1776,-3122,1458,1704,29491,
            1704,1458,-3122,1776,1104,-2690,1650,724,-2070,
            1348,392,-1380,932,158,-740,490,33,-236,105,0,
            88,-165,20,239,-299,53,257,-312,72,200,-245,67,
            119,-147,44,50,-58,16,8,0,-6,-9,23,-14};
```

图 4.5 带阻 FIR 滤波器的系数（bs2700.cof）

在第二个循环中，更新延时的抽样，计算$n+1$时刻的输出$y(n)$，即$y(n+1)$。新采集的输入抽样始终在（本例中）抽样缓冲区的开始位置单元。存储$x(n)$的存储单元现在存储新的抽样$x(n+1)$，由此计算出$n+1$时刻的输出$y(n+1)$，这种方法利用数据移动来更新延时抽样。

例 4.8 说明了有多少种不同存储器组织方法可用于延时抽样值与滤波器系数的存储更新，以及和卷积等式在同一个循环内延时抽样的更新方法。我们还介绍了使用带指针的环形缓冲区更新延时抽样，替代数据移动的方法。输出在送到编解码器 DAC 之前（右移 15 位），这样实现输出的定标，这种方法也适合于定点实现使用。

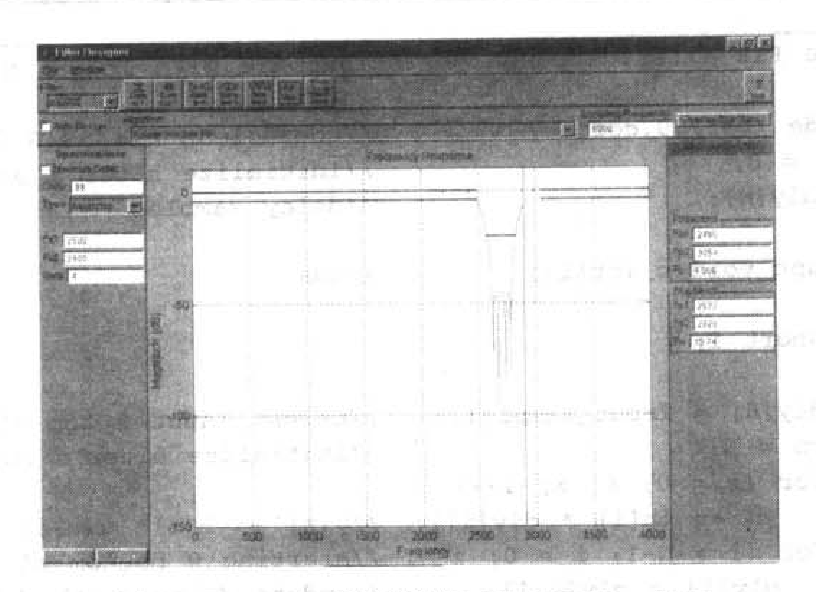


图 4.6 MATLAB 滤波器设计工具 SPTOOL，显示中心频率为 2700 Hz 的带阻滤波器的特性

中心频率为 2700 Hz 的带阻滤波器

建立并运行工程 FIR，输入一个正弦信号，并使频率在 2700 Hz 上下稍微变化，检验在 2700 Hz 时输出是最小的。

图 4.7 给出了该工程的 CCS 窗口，它显示了采用 128 点 FFT 得到滤波器系数 h（见例 1.3，起始地址为 h）的幅频特性，同时也显示了中心频率在 2700 Hz 的 FIR 带阻滤波器的特性。该图还显示了 CCS 时域图，即滤波器的冲激响应。

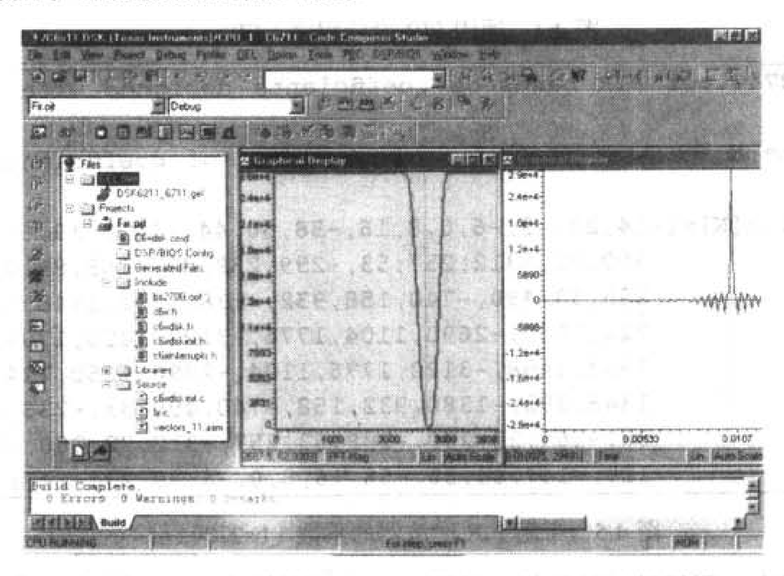


图 4.7 CCS 图：带阻滤波器系数的幅频特性（FFT）和冲激响应

用噪声作为输入可以检验带阻滤波器的输出频率响应。如后面所介绍的，可用第 2 章中产生的伪随机噪声序列或另一个噪声源（见附录 D）作为 FIR 滤波器的输入。图 4.8 给出了一个实时实现且在 2700 Hz 处幅度凹陷的滤波器幅频特性图。该图是通过将分析仪产生的噪声作为输入，然后再用 HP3561A 动态信号分析仪得到的。在大约 3500 Hz 处的滚降是由于编译码上的抗混叠低通滤波器的影响所造成的。

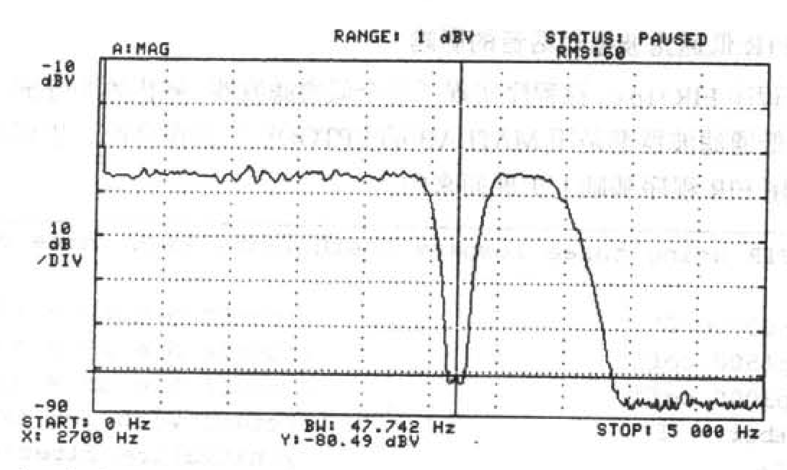


图 4.8　中心频率为 2700 Hz 的带阻滤波器的频率响应（该图用信号分析仪得到）

中心频率为 1750 Hz 的带通滤波器

在 CCS 中，编辑 FIR.c 程序，用系数文件 bp1750.cof 替代文件 bs2700.cof，文件 bp1750.cof 表示中心频率为 1750 Hz 的带通 FIR 滤波器，如图 4.9 所示。该滤波器是用 MATLAB 的滤波器工具 SPTOOL 设计的（见附录 D）。选择菜单 incremental Build，新的系数文件 bp1750.cof 就自动添加到工程了，再运行和检验中心频率为 1750 Hz 的带通 FIR 滤波器的特性。图 4.10 给出了用 HP 信号分析仪得到的滤波器输出频率响应的实时图形。

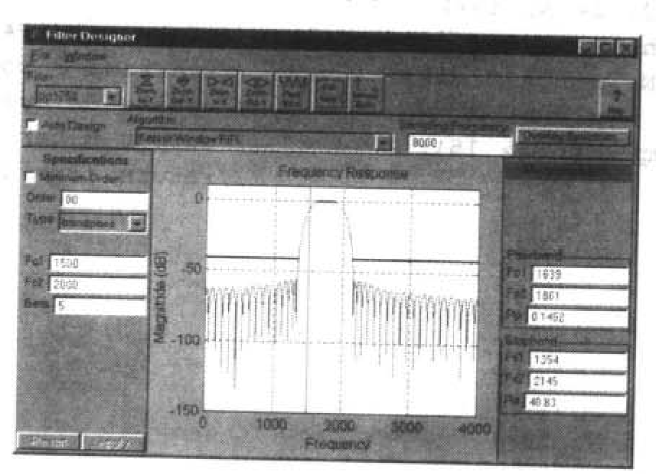


图 4.9　MATLAB SPTOOL 工具设计的中心频率为 1750 Hz 的 FIR 带通滤波器特性

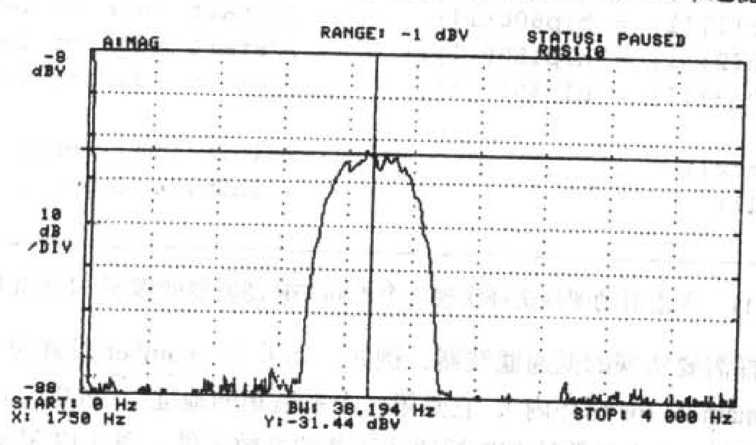


图 4.10　中心频率为 1750 Hz 的带通滤波器的输出频率响应，该图是用信号分析仪得到的

例 4.2　三个 FIR 低通滤波器对话音的影响

图 4.11 给出了程序 FIR31p.c，该程序实现了三个低通滤波器，截止频率分别为 600 Hz，1500 Hz 和 3000 Hz。三个低通滤波器都是用 MATLAB 的 SPTOOL 工具设计的，生成三组相应的系数。该例是在例 4.1 通用 FIR 程序基础上扩展而来的。

```
//FIR3LP.c FIR using three lowpass coefficients with three different BW

#include "lp600.cof"                  //coeff file LP @ 600 Hz
#include "lp1500.cof"                 //coeff file LP @ 1500 Hz
#include "lp3000.cof"                 //coeff file LP @ 3000 Hz
short LP_number = 1;                  //start with 1st LP filter
int   yn = 0;                         //initialize filter's output
short dly[N];                         //delay samples
short h[3][N];                        //filter characteristics 3xN

interrupt void c_int11()              //ISR
{
      short i;

      dly[0] = input_sample();        //newest input @ top of buffer
      yn = 0;                         //initialize filter output
      for (i = 0; i< N; i++)
         yn +=(h[LP_number][i]*dly[i]); //y(n) += h(LP#,i)*x(n-i)
      for (i = N-1; i > 0; i--)       //starting @ bottom of buffer
         dly[i] = dly[i-1];           //update delays with data move
      output_sample(yn >> 15);        //output filter
      return;                         //return from interrupt
}

void main()
{
      short i;

      for (i=0; i<N; i++)
         {
           dly[i] = 0;                //init buffer
           h[1][i] = hlp600[i];       //start addr of LP600 coeff
           h[2][i] = hlp1500[i];      //start addr of LP1500 coeff
           h[3][i] = hlp3000[i];      //start addr of LP3000 coeff
         }
      comm_intr();                    //init DSK, codec, McBSP
      while(1);                       //infinite loop
}
```

图 4.11　点击滑动图标选择实现三个不同 FIR 滤波器的程序（FIR3LP.c）

LP_number 选择需要实现的低通滤波器，例如，如果 LP_number 设置为 1，则 h[1][*i*]就等于 hlp600[*i*]（在函数 main 的 for 循环内），它是第一个系数集的地址。LP600.cof 是使用凯塞窗、截止频率为 600 Hz，具有 81 个系数的 FIR 低通滤波器的系数文件。图 4.12 显示了该系数文件（其

他两个系数集在辅助材料中)，然后实现该滤波器。LP_number 可以改为 2 或 3 来分别实现 1500 Hz 或 3000 Hz 的低通滤波器。利用 GEL 文件 FIR3LP.gel（如图 4.13 所示），可以把 LP_number 从 1 变到 3，点击 Filter 下的箭头图标从而选择实现三个不同的滤波器。

建立工程 FIR3LP，用.wav 文件 TheForce.wav（在辅助材料中）作为输入（参见附录 D），观察三个低通滤波器对输入话音的影响。选择 600 Hz 较低的带宽，使用第一个系数集，语音信号高于 600 Hz 的频率分量被抑制掉了，把输出连接到喇叭或频谱分析仪上，检查一下结果如何？观察三个不同 FIR 滤波器的带宽。

```
//LP600.cof FIR lowpass filter coefficients using Kaizer window

#define N 81          //length of filter

short hlp600[N] = {0,-6,-14,-22,-26,-24,-13,8,34,61,80,83,63,19,-43,-113,
-171,-201,-185,-117,0,146,292,398,428,355,174,-99,-416,-712,-905,-921,
-700,-218,511,1424,2425,3391,4196,4729,4915,4729,4196,3391,2425,1424,
511,-218,-700,-921,-905,-712,-416,-99,174,355,428,398,292,146,0,-117,
-185,-201,-171,-113,-43,19,63,83,80,61,34,8,-13,-24,-26,-22,-14,-6,0};
```

图 4.12　截止频率为 600 Hz 的 FIR 低通滤波器系数文件（LP600.cof）

```
/*FIR3LP.gel Gel file to step through 3 different LP filters*/

menuitem "Filter Characteristics"

slider Filter(1,3,1,1,filterparameter)  /*from 1 to 3,incr by 1*/
{
      LP_number = filterparameter;      /*for 3 LP filters*/
}
```

图 4.13　选择三个 FIR 低通滤波器系数的 GEL 文件（FIR3LP.gel）

例 4.3　低通、高通、带通和带阻 4 种不同滤波器的实现

该例与例 4.2 类似，说明使用 GEL（滑动条）文件选择 4 种不同类型的滤波器，滤波器是使用 MATLAB 的 SPTOOL 工具来设计的，每个滤波器有 81 个系数（在辅助材料中）。这 4 个系数文件分别是：

1. lp1500.cof，带宽为 1500 Hz 的低通滤波器。
2. hp2200.cof，带宽为 2200 Hz 的高通滤波器。
3. bp1750.cof，中心频率为 1750 Hz 的带通滤波器。
4. bs790.cof，中心频率为 790 Hz 的带阻滤波器。

程序 FIR4types.c（在辅助材料中）实现了该工程。将程序 FIR3LP.c（见例 4.2）稍加修改，就能实现第四个滤波器。

建立并运行工程 FIR4types，装载 GEL 文件 FIR4types.gel（在辅助材料中），检验 4 种不同的 FIR 滤波器，该例可很容易地扩展到实现更多的 FIR 滤波器。

图 4.9 显示了中心频率为 1750 Hz 的带通 FIR 滤波器的特性，该滤波器是用 MATLAB 滤波器设计工具设计的，图 4.10 显示了用 HP 信号分析仪获得的频率响应。

例 4.4 用伪随机噪声序列作为输入的 FIR 滤波器的实现

程序 FIRPRN.c（如图 4.14 所示）实现了 FIR 滤波器，并用内部生成的伪随机噪声作为滤波器的输入。输入是例 2.16 生成的伪随机噪声序列，系数文件 BP55.cof 采用浮点数据格式，如图 4.15 所示。[滤波器开发工具包（在辅助材料中）产生浮点或十六进制格式的滤波器系数，附录 E 对此进行了介绍。] 该文件表示中心频率为 $F_s/4$、包含 55 个系数的 FIR 带通滤波器。

```
//FIRPRN.c FIR with internally generated input noise sequence

#include "bp55.cof"                  //BP @ Fs/4 coeff file in float
#include "noise_gen.h"               //header file for noise sequence
int dly[N];                          //delay samples
short fb;                            //feedback variable
shift_reg sreg;

short prn(void)                      //pseudorandom noise generation
{
 short prnseq;                       //for pseudorandom sequence

 if(sreg.bt.b0)                      //sequence {1,-1}
      prnseq = -16000;               //scaled negative noise level
 else
      prnseq = 16000;                //scaled positive noise level
 fb =(sreg.bt.b0)^(sreg.bt.b1);      //XOR bits 0,1
 fb ^=(sreg.bt.b11)^(sreg.bt.b13);   //with bits 11,13 ->fb
 sreg.regval<<=1;                    //shift register 1 bit to left
 sreg.bt.b0 = fb;                    //close feedback path

 return prnseq;                      //return sequence
}

interrupt void c_int11()             //ISR
{
      int i;
      int yn = 0;                    //initialize filter's output

      dly[0] = prn();                //input noise sequence
      for (i = 0; i< N; i++)
         yn +=(h[i]*dly[i]);         //y(n)+= h(i)*x(n-i)
      for (i = N-1; i > 0; i--)      //start @ bottom of buffer
         dly[i] = dly[i-1];          //data move to update delays

      output_sample(yn);             //output filter
      return;                        //return from interrupt
}

void main()
{
      short i;
```

```
    sreg.regval = 0xFFFF;          //shift register to nominal values
    fb = 1;                        //initial feedback value
    for (i = 0; i<N; i++)
       dly[i] = 0;                 //init buffer
    comm_intr();                   //init DSK, codec, McBSP
    while(1);                      //infinite loop
}
```

图 4.14 用伪随机序列作为输入的 FIR 程序（FIRPRN.c）

建立并运行工程 FIRPRN，检验生成的滤波器是否是一个中心频率为 2 kHz 的 FIR 带通滤波器。为了检验噪声序列的输出，调用 output_sample 函数时，用输出 dly[0]代替 yn 即可。

```
//bp55.cof Coefficients for bandpass FIR filter centered @ Fs/4

#define N 55                       //number of coefficients

float h[N]=
{1.7619E-017, 7.0567E-003, 2.2150E-018,-1.0962E-002, 4.0310E-017,
 1.3946E-002, 7.1787E-018,-1.4588E-002, 3.9928E-017, 1.1474E-002,
 5.9881E-018,-3.5159E-003,-6.6174E-018,-9.7476E-003,-1.7919E-017,
 2.7932E-002,-9.4329E-017,-4.9740E-002, 3.3834E-017, 7.3066E-002,
-3.6228E-017,-9.5284E-002, 3.2194E-017, 1.1365E-001,-2.2165E-017,
-1.2576E-001, 7.8980E-018, 1.3000E-001, 7.8980E-018,-1.2576E-001,
-2.2165E-017, 1.1365E-001, 3.2194E-017,-9.5284E-002,-3.6228E-017,
 7.3066E-002, 3.3834E-017,-4.9740E-002,-9.4329E-017, 2.7932E-002,
-1.7919E-017,-9.7476E-003,-6.6174E-018,-3.5159E-003, 5.9881E-018,
 1.1474E-002, 3.9928E-017,-1.4588E-002, 7.1787E-018, 1.3946E-002,
 4.0310E-017,-1.0962E-002, 2.2150E-018, 7.0567E-003, 1.7619E-017};
```

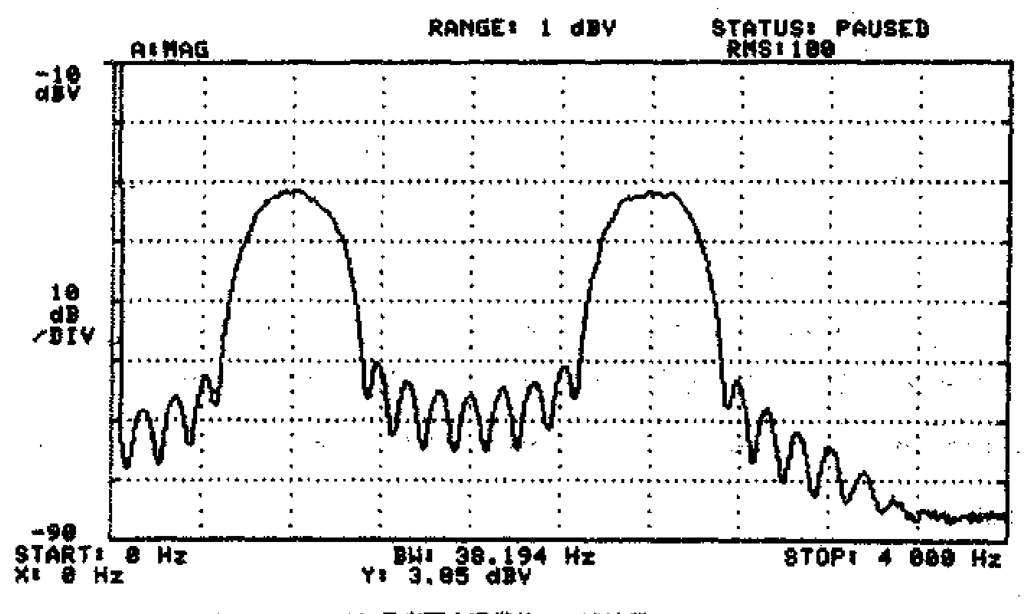
图 4.15 中心频率为$F_s/4$的 FIR 带通滤波器的浮点格式系数文件（BP55.cof）

测试不同的 FIR 滤波器

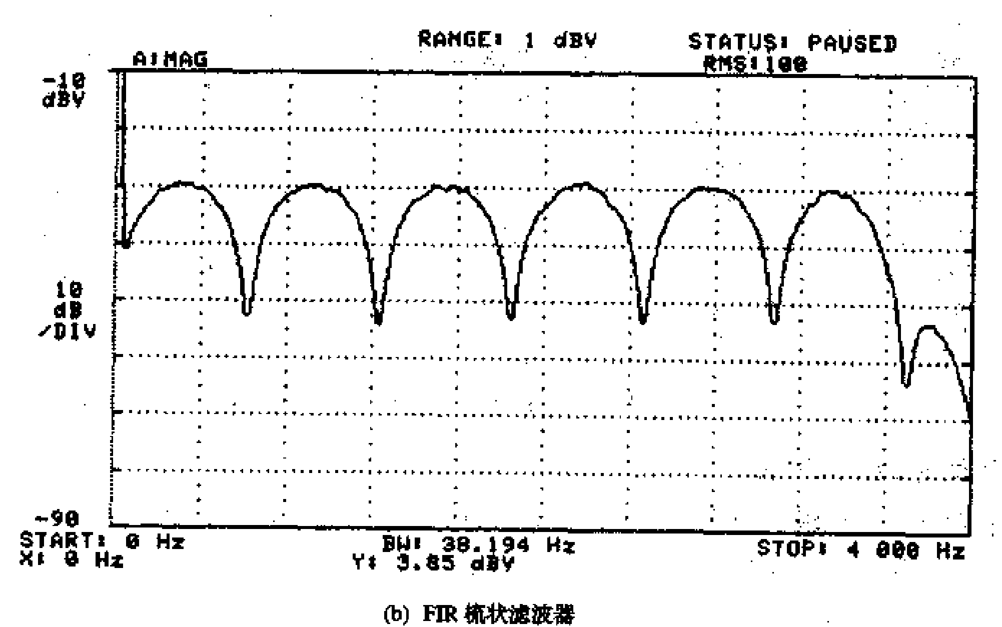
停止程序运行，编辑 C 语言源程序，添加和测试表示不同 FIR 滤波器（在辅助材料中）的系数文件，所有系数使用浮点格式，除了 comb14.cof 文件外，每个系数文件含有 55 个系数。

1、BS55.cof，中心频率为$F_s/4$的带阻滤波器。
2. BP55.cof，中心频率为$F_s/4$的带通滤波器。
3. LP55.cof，截止频率为$F_s/4$的低通滤波器。
4. HP55.cof，带宽为$F_s/4$的高通滤波器。
5. Pass2b.cof，具有两个通带的滤波器。
6. Pass3b.cof，具有三个通带的滤波器。
7. Pass4b.cof，具有四个通带的滤波器。
8. Stop3b.cof，具有三个阻带的滤波器。
9. Comb14.cof，多个陷波点的滤波器（梳状滤波器）。

图 4.16(a)显示了用系数文件 pass2b.cof 且有两个通带的 FIR 滤波器的实时输出频率响应，该滤波器是用 MATLAB 设计的。图 4.16(b)显示了系数文件为 comb14.cof 的梳状滤波器的频率响应，该图是用 HP 3561A 信号分析仪得到的。




(a) 具有两个通带的FIR滤波器



(b) FIR梳状滤波器

图4.16 用HP分析仪得到的输出频率响应

例4.5 用CCS设计具有频率响应图形的FIR滤波器

图4.17给出了程序FIRbuf.c，该文件实现了一个FIR滤波器，并将滤波器的输出存入缓冲区中，然后用CCS画出滤波器频率响应的FFT幅度谱。例4.1介绍了使用通用程序以及要求的滤波器幅度特性文件实现FIR滤波器的过程。例1.2介绍了怎样把输出保存到缓冲区，以便在CCS运行程序内画出响应的图形。程序FIRbuf.c基于前面的两个例子，系数文件bp41.cof表示中心频率在1 kHz、具有41个系数的FIR带通滤波器，输出缓冲区的大小是1024。

```
//Firbuf.c FIR filter with output in buffer plotted with CCS

#include "bp41.cof"                     //BP @ 1 kHz coefficient file

int yn = 0;                             //initialize filter's output
short dly[N];                           //delay samples
short buffercount = 0;                  //init buffer count
const short bufferlength = 1024;        //buffer size
short yn_buffer[1024];                  //output buffer

interrupt void c_int11()                //ISR
{
      short i;

      dly[0] = input_sample();          //newest input @ top of buffer
      yn = 0;                           //initialize filter's output
      for (i = 0; i< N; i++)
         yn +=(h[i]*dly[i]) >> 15;      //y(n)+=h(i)*x(n-i)
      for (i = N-1; i > 0; i--)         //start @ bottom of buffer
         dly[i] = dly[i-1];             //data move to update delays

      output_sample(yn);                //output filter

      yn_buffer[buffercount] = yn;      //filter's output into buffer
      buffercount++;                    //increment buffer count
      if(buffercount==bufferlength)     //if buffer count = size
         buffercount = 0;               //reinitialize buffer count
      return;                           //return from interrupt
}

void main()
{
     comm_intr();                       //init DSK, codec, McBSP
     while(1);                          //infinite loop
}
```

图 4.17　滤波器输出保存到存储器的 FIR 程序（FIRbuf.c）

建立工程 FIRbuf，检验输出是否是中心频率为 1 kHz 的带通滤波器，停止处理器的运行。

用噪声作为滤波器的输入，利用 CCS 画出输出频率响应图形。用共享软件 Goldwave 生成包括噪声在内的不同信号，使用声卡产生模拟输出信号（参见附录 E）。声卡输出由 Goldwave 产生的噪声，输出信号可作为 DSK 的输入。

选择菜单 View→Graph→Time/Frequency，为下列条目设置适当的值：

1. Display type : FFT magnitude
2. Start address: yn_buffer
3. Acquisition buffer size: 1024
4. FFT frame size: 1024

5．FFT order：10
6．DSP data type: 16-bit signed integer
7．Sampling rate: 8000 Hz

其他部分使用默认值，FFT 的阶数是 M，其中2^M = FFT帧长。运行程序，检验图 4.18 画出的滤波器输出频率响应。

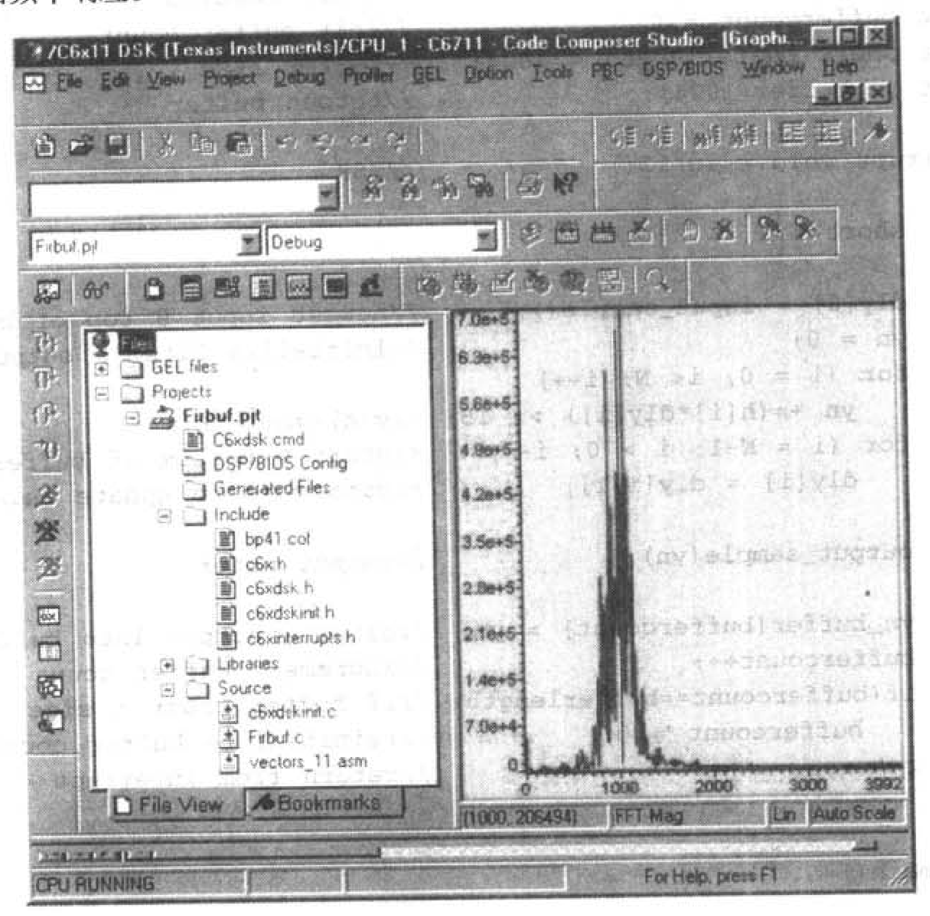


图 4.18　工程 FIRbuf 用外部噪声作为滤波器输入，CCS 画出的 1 kHz 带通 FIR 滤波器输出频率响应图形

例 4.6　用内部产生的伪随机噪声作为滤波器输入，并将输出保存到存储器的 FIR 滤波器

该例建立在例 2.16 程序 noise_gen 和例 4.5 程序 FIRbuf 的基础上，前者产生伪随机噪声序列，后者实现 FIR 滤波器，并将滤波器输出也保存到存储缓冲区。图 4.19 给出了程序 FIRPRNbuf.c，它实现了该例的工程。

滤波器的输入是软件产生的噪声序列，用 dly[0]作为最新的噪声序列，系数文件 BP41.cof 和例 4.5 一样，表示含有 41 个系数的 FIR 带通滤波器。

建立并运行工程 FIRPRNbuf，检验 1 kHz 带通滤波器的输出频率响应。Goldwave 软件也可以当成粗略的谱分析仪，得到滤波器的频率响应（DSK 的输出连接到声卡的输入）。

使用 CCS，检验如图 4.20 所示的 1024 点的 FFT 幅度谱图。输出缓冲区的地址是 yn_buffer，图 4.21 给出了中心频率为 $F_s/8$ 的 FIR 带通滤波器的频率响应，该图是由 HP 分析仪获得的。

```
//FIRPRNbuf.c FIR filter with input noise sequence & output in buffer

#include "bp41.cof"                  //BP @ 1 kHz coefficient file
#include "noise_gen.h"               //header file for noise sequence
int yn = 0;                          //initialize filter's output
short dly[N];                        //delay samples
short buffercount = 0;               //init buffer count
const short bufferlength = 1024;     //buffer size
short yn_buffer[1024];               //output buffer
short fb;                            //feedback variable
shift_reg sreg;

short prn(void)                      //pseudorandom noise generation
{
 short prnseq;                       //for pseudorandom sequence

 if(sreg.bt.b0)                      //sequence {1,-1}
      prnseq = -8000;                //scaled negative noise level
 else
      prnseq = 8000;                 //scaled positive noise level
 fb =(sreg.bt.b0)^(sreg.bt.b1);      //XOR bits 0,1
 fb ^=(sreg.bt.b11)^(sreg.bt.b13);   //with bits 11,13 ->fb
 sreg.regval<<=1;                    //shift register 1 bit to left
 sreg.bt.b0 = fb;                    //close feedback path

  return prnseq;
}

interrupt void c_int11()             //ISR
{
      short i;

      dly[0] = prn();                //input noise sequence
      yn = 0;                        //initialize filter's output
      for (i = 0; i< N; i++)
         yn +=(h[i]*dly[i]) >>15;    //y(n)+=h(i)*x(n-i)
      for (i = N-1; i > 0; i--)      //start @ bottom of buffer
         dly[i] = dly[i-1];          //data move to update delays

      output_sample(yn);             //output filter

      yn_buffer[buffercount] = yn;   //filter's output into buffer
      buffercount++;                 //increment buffer count
      if(buffercount==bufferlength)  //if buffer count = size
         buffercount = 0;            //reinitialize buffer count
      return;                        //return from interrupt
}

void main()
{
      sreg.regval = 0xFFFF;          //shift register to nominal values
      fb = 1;                        //initial feedback value
      comm_intr();                   //init DSK, codec, McBSP
      while(1);                      //infinite loop
}
```

图 4.19　输入伪随机噪声序列，输出保存到存储器的 FIR 程序（FIRPRNbuf.c）

改变输出缓冲区，使用 yn_buffer[i] = dly[0]，将噪声序列保存在存储器中，再运行程序，画出噪声序列的 FFT 幅度谱。由于输出的结果没有平均，所以输出的谱图不够平坦。

也可以在程序里使用 output_sample(dly[0])输出噪声序列，利用有平均功能的频谱分析仪输出，检验在编解码器上抗混叠滤波器带宽 3500 Hz 内，噪声频谱是否是很平坦的（就像带宽为是 3500 Hz 的低通滤波器）。图 4.22 显示了使用 HP 频谱分析仪（具有平均功能）获得的噪声序列频谱，使用 GEL 文件产生滑动条，确定 DSK 的输出是内部产生的噪声序列 dly[0]还是滤波器的输出。

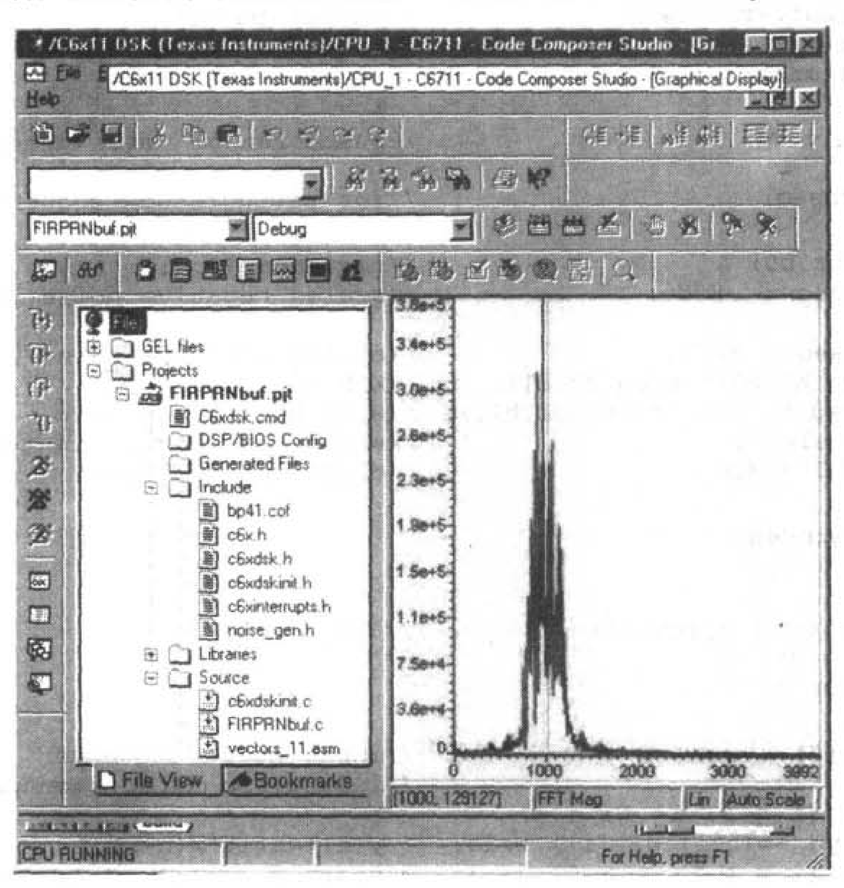


图 4.20　工程 FIRPRNbuf 利用内部产生的噪声序列作为滤波器的输入，CCS 画出的 1 kHz FIR 带通滤波器的输出频率响应

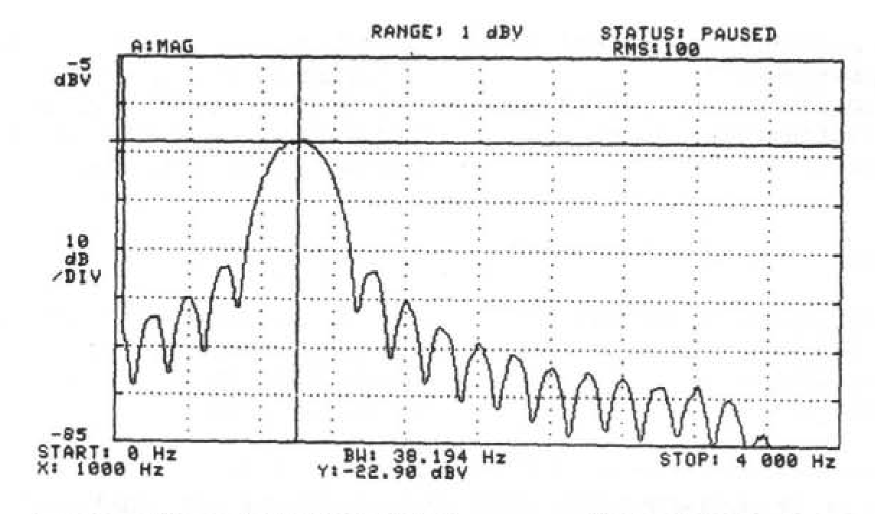


图 4.21　利用 HP 频谱分析仪获得的 1 kHz FIR 带通滤波器的频率响应

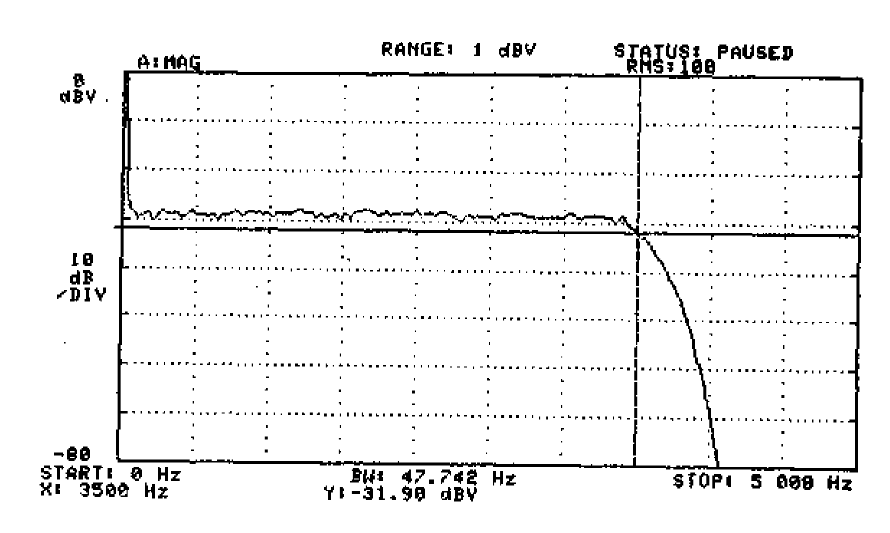


图 4.22 使用频谱分析仪获得的内部噪声序列的功率谱

例 4.7 使用两个陷波器恢复受干扰的语音

该例说明实现两个陷波器（带阻）FIR 滤波器，消除输入语音信号中不希望的两个正弦信号。用 Goldwave 软件在语音信号（辅助材料中的 TheForce.wav）中加入频率为 900 Hz 和 2700 Hz 的信号，这样得到有干扰的输入信号 corruptvoice.wav（在辅助材料中）。

图 4.23 给出了陷波器程序 NOTCH2.c，它实现了两个级联的滤波器。系数文件 BS900.cof 和 BS2700.cof（在辅助材料中）是使用 MATLAB 软件设计的，每个文件含有 89 个系数，它们放在程序 NOTCH2.c 中，分别表示中心频率在 900 Hz 和 2700 Hz 的两个 FIR 陷波器。缓冲区用于存放每个滤波器的延时抽样，第一个中心频率为 900 Hz 的陷波器的输出是第二个中心频率为 2700 Hz 的陷波器输入。

```
//Notch2.C Two FIR notch filters to remove two sinusoidal noise signals

#include "BS900.cof"                    //BS @ 900 Hz coefficient file
#include "BS2700.cof"                   //BS @ 2700 Hz coefficient file
short dly1[N]={0};                      //delay samples for 1st filter
short dly2[N]={0};                      //delay samples for 2nd filter
int y1out = 0, y2out = 0;               //init output of each filter
short out_type = 1;                     //slider for output type

interrupt void c_int11()                //ISR
{
      short i;

      dly1[0] = input_sample();         //newest input @ top of buffer
      y1out = 0;                        //init output of 1st filter
      y2out = 0;                        //init output of 2nd filter
      for (i = 0; i< N; i++)
         y1out += h900[i]*dly1[i];      //y1(n)+=h900(i)*x(n-i)

      dly2[0]=(y1out >>15);             //out of 1st filter->in 2nd filter
      for (i = 0; i< N; i++)
         y2out += h2700[i]*dly2[i];     //y2(n)+=h2700(i)*x(n-i)
```

```
        for (i = N-1; i > 0; i--)        //from bottom of buffer
          {
            dly1[i] = dly1[i-1];         //update samples of 1st buffer
            dly2[i] = dly2[i-1];         //update samples of 2nd buffer
          }

        if (out_type==1)                 //if slider is in position 1
        output_sample(dly1[0]);          //corrupted input(voice+sines)
        if (out_type==2)
        output_sample(y2out>>15);        //output of 2nd filter (voice)
        return;                          //return from ISR
}

void main()
{
        comm_intr();                     //init DSK, codec, McBSP
        while(1);                        //infinite loop
}
```

图 4.23　消除两个干扰信号的两个级联 FIR 陷波器程序（NOTCH2.c）

建立工程 NOTCH2，输入（播放）有干扰的语音文件 corruptvoice.wav，检查滑动条在第一个位置（与初始设置一样）时，选择如图 4.24 显示的有干扰的话音。该图是用 DSK 的输出作为声卡（参见附录 E）的输入，并利用 Goldwave 软件画出的。因为使用单声道信号，所以只显示了一个声道的信号（左声道）。观察分别在 900 Hz 和 2700 Hz 的两个尖峰脉冲（表示两个正弦信号）。将滑动图标放到第二个位置，检查两个不希望的信号是否被去除了。

同样，通过函数 output_sample（重建）输出 y1out，检验单独的 2700 Hz 的频率干扰输入信号的情况。

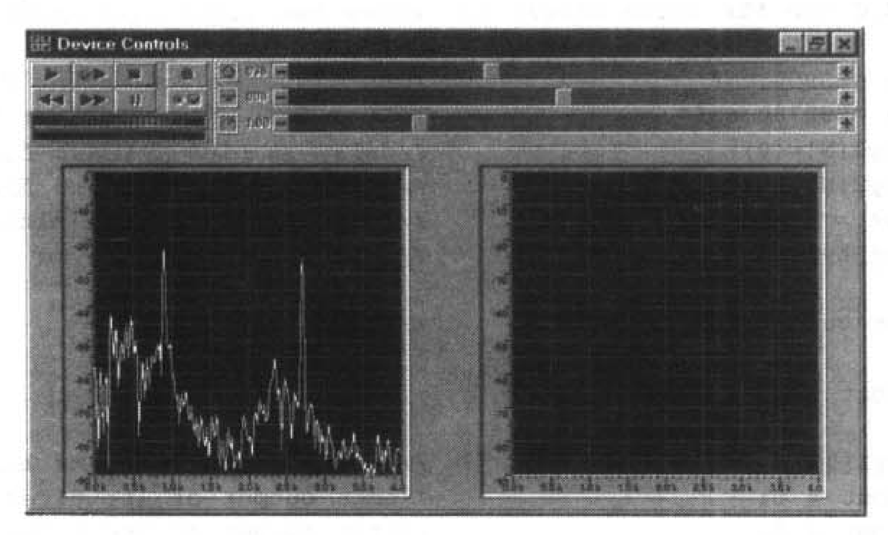


图 4.24　受 900 Hz 和 2700 Hz 两个正弦信号干扰的话音频谱（利用 Goldwave 软件得到）

例 4.8　使用 4 种不同方法实现 FIR

图 4.25 显示了程序 FIR4ways.c，该程序使用 4 种方法卷积/更新延时抽样实现 FIR 滤波器。该例扩展了使用第一种方法（方法 A）的例 4.1。在第一种方法中，使用两个 for 循环，延时抽样保存在存储器中，且最新的抽样放在缓冲区的开始，最早的抽样放在缓冲区的末尾，最新的抽样

和第一个系数利用下面的公式进行卷积：

$$y(n) = h(0)x(n) + h(1)x(n-1) + \cdots + h(N-1)x(n-(N-1))$$

```
//FIR4ways.c FIR with alternative ways of storing/updating samples

#include "bp41.cof"                 //BP @ 1 kHz coefficient file
#define METHOD 'D'                  //change to B or C or D
int yn = 0;                         //initialize filter's output
short dly[N+1];                     //delay samples array(one extra)

interrupt void c_int11()            //ISR
{
      short i;
      yn = 0;                       //initialize filter's output

#if METHOD == 'A'                   //if 1st method
      dly[0] = input_sample();      //newest sample @ top of buffer
      for (i = 0; i< N; i++)
         yn += (h[i] * dly[i]);     //y(n)=h[0]*x[n]+..+h[N-1]x[n-(N-1)]
      for (i = N-1; i > 0; i--)     //from bottom of buffer
         dly[i] = dly[i-1];         //update sample data move "down"

#elif METHOD == 'B'                 //if 2nd method
      dly[0] = input_sample();      //newest sample @ top of buffer
      for (i = N-1; i >= 0; i--)    //start @ bottom to convolve
        {
         yn += (h[i] * dly[i]);     //y=h[N-1]x[n-(N-1)]+...+h[0]x[n]
         dly[i] = dly[i-1];         //update sample data move "down"
        }
#elif METHOD == 'C'                 //use xtra memory location
      dly[0] = input_sample();      //newest sample @ top of buffer
      for (i = N-1; i>=0; i--)      //start @ bottom of buffer
        {
         yn += (h[i] * dly[i]);     //y=h[N-1]x[n-(N-1)]+...+h[0]x[n]
         dly[i+1] = dly[i];         //update sample data move "down"
        }

#elif METHOD == 'D'                 //1st convolve before loop
      dly[N-1] = input_sample();    //newest sample @ bottom of buffer
      yn = h[N-1] * dly[0];         //y=h[N-1]x[n-(N-1)] (only one)
      for (i = 1; i<N; i++)         //convolve the rest
        {
         yn +=(h[N-(i+1)]*dly[i]);  //h[N-2]x[n-(N-2)]+...+h[0]x[n]
         dly[i-1] = dly[i];         //update sample data move "up"
        }
#endif
output_sample(yn >> 15);            //output filter
return;                             //return from ISR
}

void main()
{
      comm_intr();                  //init DSK, codec, McBSP
      while(1);                     //infinite loop
}
```

图 4.25　使用 4 种方法实现卷积和更新延迟抽样的 FIR 程序（FIR4ways.c）

存储器中的每个数据“往下”移动，更新延时抽样，最新的抽样就是当前的输入抽样。为了说明第三种方法（方法C），数组的大小现设为$N+1$，而不是N；其他三种方法用N个缓冲存储单元保存延时抽样。本例中缓冲区的底部（末尾）是指第N个存储单元，而不是第$N+1$个。注意，在这种情况下，通过使用索引变量$i<N$，第$N+1$存储单元中不用的数据$x(n-N)$不被更新。

第二种方法（方法B）使用单循环实现卷积和更新抽样，卷积开始用最早的数据和最早的一个系数（按照排列顺序），抽样通过缓冲区“向上”移动，利用下面公式进行计算：

$$y(n)=h(N-1)x(n-(N-1))+h(N-2)x(n-(N-2))+\cdots+h(0)x(n)$$

更新方法类似于第一种方法。在方法B中，当$i=0$时，最新的抽样被位于抽样缓冲区之前的存储单元的无效数据替换，但该无效数据会在下一个计算$y(n)$的时间单元之前被新采集的输入抽样 dly[0]所替换。也就是说，除了$i=0$，对于所有i值都可以用一个 if 语句更新各抽样值。

第三种方法用$N+1$个存储单元来更新延时抽样，第$N+1$个存储单元中的未被使用的数据也被更新，该多余的存储单元用来避免该单元中的有效数据被覆盖。

第四种方法是在循环外计算第一个卷积表达式。在以上几种方法中，延时抽样保存在存储单元中，最新的抽样$x(n)$保存在缓冲区的开始，而最早的抽样$x(n-(N-1))$在缓冲区尾部。但是，在这种方法中，最新输入的抽样是通过 dly[N–1]得到的，因此，它位于缓冲区的尾部，更新过程将数据往上移动。

建立并运行 FIR4ways 工程，检验输出是否是中心频率为 1 kHz 的 FIR 带通滤波器，然后再依次换成其他三种方法，检验这三种方法并观察输出结果是否一样。

例 4.9　采用滤波和调制的语音扰乱器

该例说明了语音扰乱/解扰的一种方法，它使用了滤波与调制的基本算法。在例 2.14 中已介绍过调制，其中语音信号为输入，输出为扰乱的语音信号。当一个 DSK 信号输入到运行同样程序的另一个 DSK 时，扰乱的语音信号就被恢复成原始未被扰乱的语音信号。

用 16 kHz 的上抽样处理代替 AD535 编解码器的 8 kHz 处理，性能会更好，因为这样允许输入信号带宽范围更宽。

扰乱方法通常被称为频率倒置，它输入频率为 0.3~3 kHz 的音频范围信号，将其调制到载波信号上。频率倒置是通过将输入音频信号与载波相乘（调制），使其频谱产生移动并生成上、下边带，下边带代表可听到的音频部分，通过这种方法，原来低音部分变为高音，反过来也一样。

图 4.26 是扰乱方案的框图，在 A 点是 0~3 kHz 的带宽受限信号，在 B 点得到抑制载波的双边带信号，在 C 点上边带被滤掉。这种方法的优点在于实现简单，只用到简单的 DSP 算法：滤波、正弦产生/调制、上抽样（AD535 编解码器的抽样速率低）。

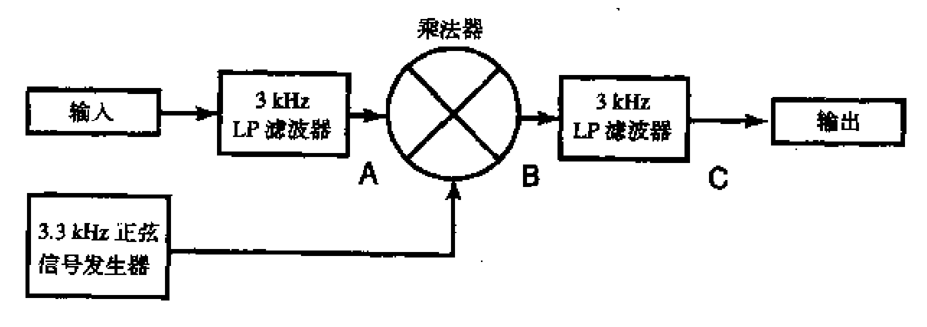


图 4.26　扰乱方案的框图

图 4.27 给出了实现该方案的程序 Scram16k.c，输入信号先通过一个低通滤波器，其输出（A

点）与 3.3 kHz 的正弦函数相乘（调制），正弦数据由缓冲区中获得，B 点调制的信号再被滤波，在 C 点只有下边带的信号。

除了 main 函数外，图 4.27 中还有三个函数，其中 filtmodfilt 调用了第一个低通滤波器，实现抗混叠滤波，低通滤波器的输出作为乘法/调制器的输入。函数 sinemod 用 3.3 kHz 正弦数据来调制（相乘）已滤波的信号，产生上、下边带信号分量，调制后的输出再经滤波，这时只有下边带信号分量被保留下来。

上抽样获得 16 kHz 抽样率是通过对数据进行两次处理，但只保留第二次结果来实现的，这样可以扰乱更宽带宽的输入信号。

低通滤波器的 114 个系数保存在一个缓冲区中，系数文件 lp114.cof 在辅助材料中。另外有两个缓冲区用来存储延迟的抽样值，抽样值在存储器中是这样存放的：

$$x(n-(N-1)), x(n-(N-2)), \cdots, x(n-1), x(n)$$

最早的数据在缓冲区开始，而最新抽样值存在缓冲区末尾（底部）。160 个数据共有 33 个周期，保存在文件 sine160.h 中（见辅助材料）。生成的信号频率为：

$$f = F_s \times 周期数/点数 = 16\,000 \times 33/160 = 3.3\ \text{kHz}$$

```
//Scram16k.c Voice scrambler/de-scrambler program

#include "sine160.h"                 //sine data values
#include "LP114.cof"                 //filter coefficient file
short filtmodfilt(short data);
short filter(short inp,short *dly);
short sinemod(short input);
static short filter1[N],filter2[N];
short input, output;

void main()
{
 short i;

 comm_poll();                        //init DSK using polling
 for (i=0; i< N; i++)
   {
    filter1[i] = 0;                  //init 1st filter buffer
    filter2[i] = 0;                  //init 2nd filter buffer
   }
 while(1)
   {
    input=input_sample();            //input new sample data
    filtmodfilt(input);              //process sample twice(upsample)
    output=filtmodfilt(input);       //and throw away 1st result
    output_sample(output);           //then output
   }
}
short filtmodfilt(short data)        //filtering & modulating
{
 data = filter(data,filter1);        //newest in ->1st filter
 data = sinemod(data);               //modulate with 1st filter out
 data = filter(data,filter2);        //2nd LP filter
 return data;
}

short filter(short inp,short *dly)   //implements FIR
{
 short i;
 int yn;
```

```
   dly[N-1] = inp;                    //newest sample @bottom buffer
   yn = dly[0] * h[N-1];              //y(0)=x(n-(N-1))*h(N-1)
   for (i = 1; i < N; i++)            //loop for the rest
     {
      yn += dly[i] * h[N-(i+1)];      //y(n)=x[n-(N-1-i)]*h[N-1-i]
      dly[i-1] = dly[i];              //data up to update delays
     }
   yn = ((yn) >>15);                  //filter's output
   return yn;                         //return y(n) at time n
  }

short sinemod(short input)            //sine generation/modulation
{
 static short i=0;
 input=(input*sine160[i++])>>11;      //(input)*(sine data)
 if(i>= NSINE) i = 0;                 //if end of sine table
 return input;                        //return modulated signal
}
```

图 4.27 语音扰乱程序（Scram16k.c）

将第一个 DSK 的最后输出当做另一个运行同样算法的 DSK 的输入，第二个 DSK 的输出就是原始未扰乱的语音信号。注意，当并口电缆（DB25）断开时，第一个 DSK 的程序仍在运行。

建立并运行工程 Scram16k，先输入 2 kHz 正弦波，测试输出是 1.3 kHz 的下边带信号（3.3 kHz－2 kHz），上边带信号（3.3 kHz＋2 kHz）被第二级低通滤波器滤掉了。

第二个 DSK 用来恢复/解扰原始信号（模拟接收机），将第一个 DSK 的输出作为第二个 DSK 的输入，并运行同样程序。这是第一个 DSK 的逆过程，得到原始未扰乱的信号。若使用 2 kHz 的信号作为输入，那么第二个 DSK 的输入就是 1.3 kHz 的扰乱信号，最后输出是 2 kHz（3.3 kHz－1.3 kHz）的原始信号，也就是下边带信号。

如果输入是频率递增的正弦扫频信号，则输出为频率递减的扫频信号，用文件 TheForce.wav 作为输入，检验扰乱和解扰方法的正确性。

通过动态改变调制频率或按照预先规定的序列加入（或去掉）载波频率，可使语音信号更难被截获。例如，第一个码不进行调制，第二个调制在频率f_{c1}，第三个则调制在频率f_{c2}。

该工程最先在 TMS320C25[49]实现，在 TMS320C31 DSK 上无需上抽样也实现了该工程。

例 4.10 下抽样的混叠效应的演示

图 4.28 给出了实现该工程的程序 aliasing.c。为了说明混叠效应，将处理速率降低一半，等效于 4 kHz 的速率。注意在 AD535 编解码器上，抗混叠和重构滤波器是固定的，不能旁路或改变它，用上抽样和低通滤波器将速率为 4 kHz 的抽样送到抽样速率为 8 kHz 的 AD535 编解码器。

建立工程文件 aliasing，读入配置选择文件 aliasing.gel（在辅助材料中），程序中将抗混叠初始值设为 0，因此将产生混叠现象。

1. 输入正弦信号，检验当输入信号频率达到 2 kHz 时，由于没有频率混叠效应，输出基本上是一个延迟程序（将输入延迟）。当输入信号频率增加到 2.5 kHz 时，检验输出是一个 1.5 kHz 的混叠信号。同样，3 kHz 和 3.5 kHz 的输入信号将分别产生 1 kHz 和 0.5 kHz 的混叠信号输出。由于 AD535 编解码器上抗混叠滤波器的作用，频率高于 3.5 kHz 的输入信号被抑制了。
2. 把滑动条改在位置 1，则在 4 kHz 下抽样时，需要抗混叠滤波器。当输入频率达到 1.8 kHz 时，输出是输入的延迟。但当输入信号频率超过 1.8 kHz 时，可看到输出衰减至 0，这是因为受 1.8 kHz 抗混叠低通滤波器的影响，它是由系数文件 lp33.cof 实现的（在辅助材料中）。

另外，也可用 Goldwave 软件播放 sweep.wav 文件，代替正弦信号作为输入进行实验（参见附录 E）。

```
//Aliasing.c illustration of downsampling, aliasing, upsampling

#include "lp33.cof"                   //lowpass at 1.8 kHz
short flag = 0;                       //toggles for 2x down-sampling
float indly[N],outdly[N];             //antialias and reconst delay lines
short i;                              //index
float yn;                             //filter output
short antialiasing = 0;               //init for no antialiasing filter

interrupt void c_int11()              //ISR
{

 indly[0]=(float)(input_sample());    //new sample to antialias filter
 yn = 0.0;                            //initialize downsampled value
 if (flag == 0)                       //discard input sample value
     flag = 1;                        //don't discard at next sampling
 else
  {
   if (antialiasing == 1)             //if antialiasing filter desired
    {                                 //compute downsampled value
     for (i = 0 ; i < N ; i++)        //using LP @ 1.8 kHz filter coeffs
        yn += (h[i]*indly[i]);        //filter is implemented using float
    }
   else                               //if filter is bypassed
        yn = indly[0];                //downsampled value is input value
   flag = 0;                          //next input value will be discarded
  }
 for (i = N-1; i > 0; i--)
       indly[i] = indly[i-1];         //update input buffer

 outdly[0] = (yn);                    //input to reconst filter
 yn = 0.0;                            //4 kHz sample values and zeros
 for (i = 0 ; i < N ; i++)            //are filtered at 8 kHz rate
       yn += (h[i]*outdly[i]);        //by reconstruction lowpass filter
 for (i = N-1; i > 0; i--)
       outdly[i] = outdly[i-1];       //update delays

 output_sample((short)(yn));          //8 kHz rate sample
 return;                              //return from interrupt
}

void main()
{
 comm_intr();                         //init DSK, codec, McBSP
 while(1);                            //infinite loop
}
```

图 4.28　演示 4 kHz 下抽样时混叠与抗混叠特性的程序（aliasing.c）

例 4.11 逆 FIR 滤波器的实现

图 4.29 给出了逆 FIR 滤波器实现程序 FIRinverse.c，FIR 滤波器的原始输入序列可以用逆 FIR 滤波器恢复，N 阶的 FIR 滤波器的传输函数是：

$$H(z)=\sum_{i=0}^{N-1}h_i z^{-i}$$

其中，h_i表示冲激响应系数。FIR 滤波器的输出序列为：

$$y(n)=\sum_{i=0}^{N-1}h_i x(n-i)=h_0x(n)+h_1x(n-1)+\cdots+h_{N-1}x(n-(N-1))$$

其中$x(n-i)$表示输入序列。用$\hat{x}(n)$作为$x(n)$的估计值，原始输入序列可用下面公式恢复出来即：

$$\hat{x}(n)=\frac{y(n)-\sum_{i=1}^{N-1}h_i\hat{x}(n-i)}{h_0}$$

```
//FIRinverse.c Implementation of inverse FIR Filter

#include "bp41.cof"            //coefficient file BP @ Fs/8
int yn;                        //filter's output
short dly[N];                  //delay samples
int out_type = 1;              //output type for slider

interrupt void c_int11()       //ISR
{
 short i;

 dly[0] = input_sample();      //newest input sample data
 yn = 0;                       //initialize filter's output

 for (i = 0; i<N; i++)
   yn += (h[i]*dly[i]);        //y(n)+=h(i)*x(n-i)
 if(out_type==1)               //if slider in position 1
    output_sample(dly[0]);     //original input
 if(out_type==2)
    output_sample(yn>>15);     //output of FIR filter
 if(out_type==3)               //calculate inverse FIR
  {
   for (i = N-1; i>1; i--)
     yn -= (h[i]*dly[i]);      //calculate inverse FIR filter
   yn = yn/h[0];               //scale output of inverse filter
   output_sample(yn>>8);       //send output of inverse filter
  }
 for (i = N-1; i>0; i--)       //from bottom of buffer
   dly[i] = dly[i-1];          //update delay samples
 return;                       //return from ISR
}

void main()
{
```

```
    comm_intr();                   //init DSK, codec, McBSP
    while(1);                      //infinite loop
   }
```

图 4.29　逆滤波器实现程序（FIRinverse.c）

建立工程 FIRinverse，用噪声做输入（由 Goldwave 软件或者噪声发生器产生，或修改程序，使用伪随机噪声序列等）。当滑动条选择在位置 1（默认位置）时，检验输出是否是输入的噪声序列；将滑动条选择在位置 2 时，检验中心频率为 1 kHz 的 FIR 带通滤波器的输出；当将滑动条选择在位置 3 时，这时进行 FIR 滤波器逆过程的计算，这样，最后得到原始输入噪声序列。

例 4.12　调用汇编函数的 C 语言实现 FIR

该例介绍使用 C 程序 FIRcasm.c（如图 4.30）调用汇编函数 FIRcasmfunc.asm（如图 4.31）实现 FIR 滤波器的方法。

```
//FIRCASM.c FIR C program calling ASM function fircasmfunc.asm

#include "bp41.cof"                  //BP @ Fs/8 coefficient file
int yn = 0;                          //initialize filter's output
short dly[N];                        //delay samples

interrupt void c_int11()             //ISR
{
      dly[N-1] = input_sample();     //newest sample @bottom buffer
      yn = fircasmfunc(dly,h,N);     //to ASM func through A4,B4,A6
      output_sample(yn >> 15);       //filter's output
      return;                        //return from ISR
}

void main()
{
      short i;

      for (i = 0; i<N; i++)
      dly[i] = 0;                    //init buffer for delays
      comm_intr();                   //init DSK, codec, McBSP
      while(1);                      //infinite loop
}
```

图 4.30　调用汇编函数实现 FIR 的 C 程序（FIRcasm.c）

建立并运行工程 FIRcasm，检验输出是 1 kHz 的 FIR 带通滤波器。建立两个缓冲区：dly 存放抽样数据，h 存放滤波器系数。当每一次中断发生时，就会采集到一个新的数据，并将其存放到缓冲区 dly 的末尾（高地址存储器）。表 4.3 列出了抽样数据与滤波器系数在存储器中的排列方法，即最早的抽样数据在缓冲区的开始，最新数据在缓冲区末尾。系数排列的顺序是 $h(0)$在缓冲区的开始，$h(N-1)$在缓冲区末尾。

抽样数据缓冲区、滤波器系数缓冲区的地址及缓冲区的大小分别通过寄存器 A4、B4 和 A6 传给汇编函数，因为每个存储单元中数据是按字节存放的，通过寄存器 A6 传送的缓冲区大小都增加了一倍。指针 A4 和 B4 按两字节（两个存储单元）增 1 或减 1，系数缓冲区的末尾地址存放在寄存器 B4 中，末尾地址是$2N-1$。

表 4.3 程序 FIRcasm 中滤波器系数和抽样数据在存储器中的组织方式

系数	抽样 n 时刻		抽样 n+1 时刻
$h(0)$	A4→$x(n-(N-1))$		A4→$x(n-(N-2))$
$h(1)$	$x(n-(N-2))$		$x(n-(N-3))$
$h(2)$	$x(n-(N-3))$		$x(n-(N-4))$
⋮	⋮		⋮
$h(N-2)$	$x(n-1)$		$x(n-1)$
B4→$h(N-1)$	$x(n)$	←最新数据→	$x(n)$

两条 LDH 指令读取寄存器 A4 和 B4 中存储的地址所指的存储单元的内容，分别取出最早的抽样$x(n-(N-1))$和$h(N-1)$，然后 A4 再递减指向$x(n-(N-2))$，B4 递减指向$h(N-2)$。第一次累加运算后，最早抽样值被更新。寄存器 A4 指定的单元内容被传送到 A7，然后再保存在前面的存储单元中，这是因为 A4 的值虽减小，但并没有改变指向保存最早抽样值的存储单元，因此最早抽样值$x(n-(N-1))$被$x(n-(N-2))$替代（更新），抽样值的更新是为下一个时间单元做准备的。当正在计算 n 时刻输出时，抽样值就为$n+1$时刻做准备。n 时刻滤波器的输出为：

$$y(n)=h(N-1)x(n-(N-1))+h(N-2)x(n-(N-2))+\cdots+h(1)x(n-1)+h(0)x(n)$$

```
;FIRCASMfunc.asm ASM function called from C to implement FIR
;A4 = Samples address, B4 = coeff address, A6 = filter order
;Delays organized as:x(n-(N-1))...x(n);coeff as h[0]...h[N-1]

            .def    _fircasmfunc
_fircasmfunc:                       ;ASM function called from C
            MV      A6,A1           ;setup loop count
            MPY     A6,2,A6         ;since dly buffer data as byte
            ZERO    A8              ;init A8 for accumulation
            ADD     A6,B4,B4        ;since coeff buffer data as byte
            SUB     B4,1,B4         ;B4=bottom coeff array h[N-1]
loop:                               ;start of FIR loop
            LDH     *A4++,A2        ;A2=x[n-(N-1)+i] i=0,1,...,N-1
            LDH     *B4--,B2        ;B2=h[N-1-i] i=0,1,...,N-1
            NOP     4
            MPY     A2,B2,A6        ;A6=x[n-(N-1)+i]*h[N-1-i]
            NOP
            ADD     A6,A8,A8        ;accumlate in A8

            LDH     *A4,A7          ;A7=x[(n-(N-1)+i+1]update delays
            NOP     4               ;using data move "up"
            STH     A7,*-A4[1]      ;-->x[(n-(N-1)+i] update sample
            SUB     A1,1,A1         ;decrement loop count
     [A1]   B       loop            ;branch to loop if count # 0
            NOP     5

            MV      A8,A4           ;result returned in A4
            B       B3              ;return addr to calling routine
            NOP     5
```

图 4.31 C 调用的 FIR 汇编函数（FIRcasmfunc.asm）

循环进行 41 次，依次计算出时刻 n，$n+1$和$n+2$时的输出值，并为下一时间单元更新抽样值。最新的抽样值在此过程中也被替换，无效数据驻留在缓冲区尾部后的存储单元，不过在每个时间单元，编解码器上的 ADC 获得的最新抽样值会将它覆盖掉，从而弥补上述问题。

在寄存器 A8 中进行累加，每个时间单元得到的结果被送到寄存器 A4，再返回给调用函数，调用函数的地址在寄存器 B3 中。

观察存储器中的抽样值更新情况

1. 选择菜单 Select→View→Memory，使用 16 位十六进制数据格式，起始地址为 dly，各时刻抽样值以字节形式保存在 82 个（而不是 41 个）存储单元中。滤波器系数保存在缓冲区 h 中，也占用 82 个存储单元。检验以 16 位或半个字形式保存在系数缓冲区中的系数，点击右键 Memory 窗口，取消选择"Float in Main Window"选项，这时可更好地观察源程序和存储器的内容。
2. 选择菜单 Select→View→Mixed C/ASM，在函数 FIRcasmfunc.asm 内下面的指令行设置断点：

```
MV      A8,A4
```

   也可以在该指令行双击或用右键点击设置断点。
3. 选择菜单 Select→Debug→Animate（第 1 章已介绍过），每个时间单元程序执行到断点处停止，这时观察抽样缓冲区最下面存储单元，检验在缓冲区末尾的最新抽样数据，该数据然后会向缓冲区上面移动。观察一段时间抽样值的更新情况，即存储器缓冲区中的每个数据向上移动，同时可以观察到寄存器 A4（指针）按 2 递增（两个字节）和 B4 按 2 递减。

例 4.13　调用快速汇编函数的 C 语言实现 FIR

该例与例 4.12 中使用相同的 C 调用函数（FIRcasm.c），它调用在文件 FIRcasmfuncfast（而不是 FIRcasmfunc）内的汇编函数 Fircasmfunc.asm（如图 4.32 所示）。

该函数使用了并行指令，并对指令进行了重新组织，因此比例 4.12 中的函数执行速度快。该函数利用了两条并行指令：LDH/LDH 和 SUB/LDH。

1. NOP 指令由 19 个减到 11 个。
2. 将使循环计数减小的 SUB 指令移到程序上方。
3. 改变了一些指令的次序，"填补"一些 NOP 指令的时隙。

例如，因为转移指令有 5 个延迟时隙，所以加法指令 ADD 在 A8 累加执行后，执行条件分支指令。由于对指令进行进一步排序，一些附加的变动将加快程序的运行速度，但同时程序也将变得不容易理解了。

建立工程 FIRcasmfast，连接器将执行名为 FIRcasmfast.out 的可执行文件，输出结果是例 4.12 中的 1 kHz 带通滤波器。

例 4.14　用 C 语言调用汇编函数并使用循环缓冲区实现 FIR

C 程序 FIRcirc.c（如图 4.33 所示）调用了汇编函数 FIRcircfunc.asm（如图 4.34 所示），并利用循环缓冲区来实现 FIR 滤波器，该例扩展了例 4.13。文件 bp1750.cof 中的系数是利用 MATLAB 设计的，窗函数是凯塞窗，文件内有 128 个系数，表示中心频率为 1750 Hz 的 FIR 带通滤波器。图 4.35 显示了滤波器的特性，该滤波器是用 MATLAB 的滤波器设计工具 SPTOOL 实现的（见附录 D）。

```
;FIRCASMfuncfast.asm C-called faster function to implement FIR
            .def    _fircasmfunc
_fircasmfunc:                       ;ASM function called from C
            MV      A6,A1           ;setup loop count
            MPY     A6,2,A6         ;since dly buffer data as byte
            ZERO    A8              ;init A8 for accumulation
            ADD     A6,B4,B4        ;since coeff buffer data as byte
            SUB     B4,1,B4         ;B4=bottom coeff array h[N-1]
loop:                               ;start of FIR loop
            LDH     *A4++,A2        ;A2=x[n-(N-1)+i] i=0,1,...,N-1
    ||      LDH     *B4--,B2        ;B2=h[N-1-i] i=0,1,...,N-1
            SUB     A1,1,A1         ;decrement loop count
    ||      LDH     *A4,A7          ;A7=x[(n-(N-1)+i+1]update delays
            NOP     4
            STH     A7,*-A4[1]      ;-->x[(n-(N-1)+i] update sample
    [A1]    B       loop            ;branch to loop if count # 0
            NOP     2
            MPY     A2,B2,A6        ;A6=x[n-(N-1)+i]*h[N-1-i]
            NOP
            ADD     A6,A8,A8        ;accumlate in A8

            B       B3              ;return addr to calling routine
            MV      A8,A4           ;result returned in A4
            NOP     4
```

图 4.32 具有较快执行速度的 C 调用汇编函数（FIRcasmfuncfast.asm）

```
//FIRcirc.c C program calling ASM function using circular buffer

#include "bp1750.cof"                   //BP at 1750 Hz coeff file
int yn = 0;                             //init filter's output

interrupt void c_int11()                //ISR
{
 short sample_data;

 sample_data = input_sample();          //newest input sample data
 yn = fircircfunc(sample_data,h,N);     //ASM func passing to A4,B4,A6
 output_sample(yn >> 15);               //filter's output
 return;                                //return to calling function
}

void main()
{
 comm_intr();                           //init DSK, codec, McBSP
 while(1);                              //infinite loop
}
```

图 4.33 调用汇编函数并使用循环缓冲区的 C 程序（FIRcirc.c）

为了代替移动数据更新延迟的抽样，例中使用了数据指针。地址模式寄存器（AMR）的最低 16 位设置为：

0x0040 = 0000 0000 0100 0000。

本例把寄存器 A7 的模式设置为循环缓冲区指针寄存器。AMR 的高 16 位通过 $N=0x0007$ 设

置，将块 BK0 作为循环缓冲区。缓冲区的大小是$2^{N+1}=256$。本例中循环缓冲区只用于存放延迟抽样数据。也可从使用第二个循环缓冲区来存放滤波器系数。例如使用：

```
0x0140 = 0000 0001 0100 0000
```

将选择 B4 和 A7 两个指针。

```
;FIRcircfunc.asm ASM function called from C using circular addressing
;A4=newest sample, B4=coefficient address, A6=filter order
;Delay samples organized: x[n-(N-1)]...x[n]; coeff as h(0)...h[N-1]

            .def    _fircircfunc
            .def    last_addr
            .def    delays
            .sect   "circdata"      ;circular data section
            .align 256              ;align delay buffer 256-byte boundary
delays      .space 256              ;init 256-byte buffer with 0's
last_addr   .int    last_addr-1     ;point to bottom of delays buffer
            .text                   ;code section
_fircircfunc:                        ;FIR function using circ addr
            MV      A6,A1           ;setup loop count
            MPY     A6,2,A6         ;since dly buffer data as byte
            ZERO    A8              ;init A8 for accumulation

            ADD     A6,B4,B4        ;since coeff buffer data as bytes
            SUB     B4,1,B4         ;B4=bottom coeff array h[N-1]

            MVKL    0x00070040,B6   ;select A7 as pointer and BK0
            MVKH    0x00070040,B6   ;BK0 for 256 bytes (128 shorts)

            MVC     B6,AMR          ;set address mode register AMR

            MVK     last_addr,A9    ;A9=last circ addr(lower 16 bits)
            MVKH    last_addr,A9    ;last circ addr (higher 16 bits)
            LDW     *A9,A7          ;A7=last circ addr
            NOP     4
            STH     A4,*A7++        ;newest sample-->last address

loop:                               ;begin FIR loop
            LDH     *A7++,A2        ;A2=x[n-(N-1)+i] i=0,1,...,N-1
    ||      LDH     *B4--,B2        ;B2=h[N-1-i] i=0,1,...,N-1
            SUB     A1,1,A1         ;decrement count

    [A1]    B       loop            ;branch to loop if count # 0
            NOP     2
            MPY     A2,B2,A6        ;A6=x[n-(N-1)+i]*h[N-1+i]
            NOP
            ADD     A6,A8,A8        ;accumulate in A8

            STW     A7,*A9          ;store last circ addr to last_addr
            B       B3              ;return addr to calling routine
            MV      A8,A4           ;result returned in A4
            NOP     4
```

图 4.34　使用循环缓冲区更新抽样的 C 调用汇编函数（FIRcircfunc.asm）

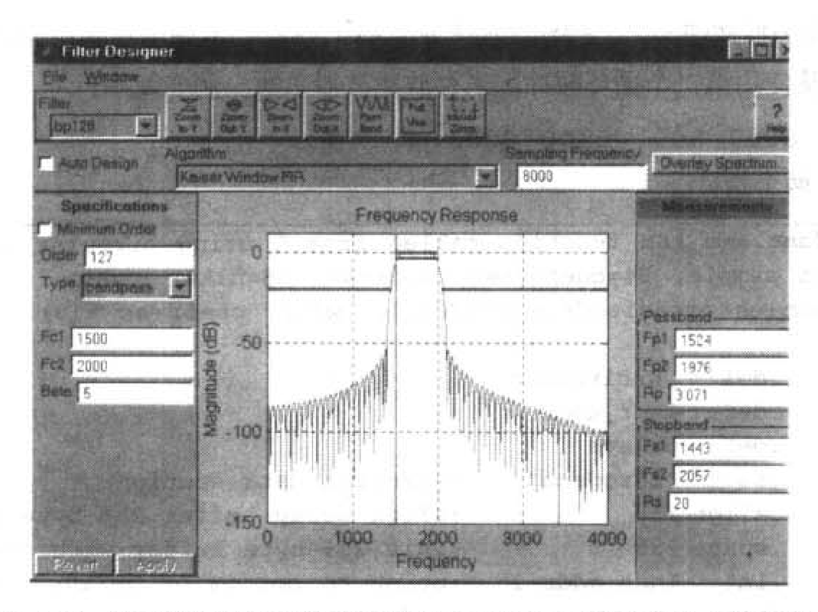


图 4.35 用 MATLAB 滤波器设计工具 SPTOOL 设计的中心频率为 1750 Hz，包含 128 个系数的 FIR 带通滤波器的频率特性图

在 C 程序中，程序行里的汇编指令可和 asm 语句一起使用，例如：

```
asm(" MVK  0x0040,B6")
```

注意在第一个引号后有空格，指令并不是从第一列开始的。循环寻址模式避免了数据移动来更新抽样值，因为使用指针移动可实现同样的较高运行速度。

寄存器指针 A7 最初指向抽样缓冲区的最后一个地址单元，暂时只考虑抽样缓冲区，因为它是循环的。

1. n 时刻。在 n 时刻，A7 指向缓冲区末尾，即最新抽样值的存储单元，然后指针逐步指向缓冲区开始，如表 4.4 所示，接着执行程序的循环部分，并计算：

$$y(n)=h(N-1)x(n-(N-1))+h(N-2)x(n-(N-2))+\cdots$$
$$+h(1)x(n-1)+h(0)x(n)$$

计算完最后一项乘积 $h(0)x(n)$ 后，A7 再增加并指向缓冲区开始的存储单元，n 时刻滤波器的输出结果返回给调用函数。在每个时间单元循环开始执行前，寄存器 A7 作为指针始终指向最新抽样的存储单元。在每个时刻 $n, n+1, n+2, \cdots$ 新的抽样值通过 A4 传给汇编函数，而 A4 保存在 A7 中，A7 总是指向最后一个地址单元。

表 4.4 使用循环缓冲存储器的系数和抽样的内存组织

系数	抽样					
		n 时刻		$n+1$ 时刻		$n+2$ 时刻
$h(0)$	A7	$\to x(n-(N-1))$	最新	$\to x(n+1)$	最新	$x(n+1)$
$h(1)$		$x(n-(N-1))$	A7	$\to x(n-(N-2))$	A7	$\to x(n+2)$
$h(2)$		$x(n-(N-1))$		$x(n-(N-3))$		$\to x(n-(N-3))$
$\vdots$		$\vdots$		$\vdots$		$\vdots$
$h(N-2)$		$x(n-1))$		$x(n-1)$		$x(n-1)$
$h(N-1)$	最新	$\to x(n)$		$x(n)$		$x(n)$

2. $n+1$时刻。在$n+1$时刻，最新抽样$x(n+1)$通过 A4 传递给汇编函数，STH 指令将该抽样值保存到 A7 所指向的缓冲存储单元，A7 现在是指向缓冲区的开始单元，然后指针 A7 增加，指向存储$x(n-(N-2))$的地址单元，如表 4.4 所示，这时输出为：

$$y(n+1)=h(N-1)x(n-(N-2))+h(N-2)x(n-(N-3))+\cdots$$
$$+h(1)x(n)+h(0)x(n+1)$$

最后一项乘积始终是$h(0)$和最新抽样的乘积。

3. $n+2$时刻。在$n+2$时刻，滤波器输出为：

$$y(n+2)=h(N-1)x(n-(N-3))+h(N-2)x(n-(N-4))+\cdots$$
$$+h(1)x(n+1)+h(0)x(n+2)$$

注意，在每个时间单元，新得到的抽样会将以前旧的抽样覆盖掉。在$n, n+1, \cdots$时刻，由汇编函数计算出滤波器的输出，然后再将结果送到 C 调用函数，这里，每个抽样周期会得到一个新抽样。

像例 4.13 一样，条件转移指令移动到了上面。当 A8 中的累加指令 ADD 完成后，开始运行指向循环的分支指令（因为该指令有 5 个延时时隙）。在处理到一个缓冲区末尾时，可以先将 AMR 的内容保存起来，在使用它之前，再用一对含有寄存器 B 的 MVC 指令：MVC AMR，Bx 和 MVC Bx，AMR 把 AMR 的内容恢复出来。

建立并运行工程 FIRcirc，检验中心频率为 1750 Hz 的 FIR 带通滤波器。再停止运行程序，复位、重新装载程序。

在汇编函数 FIRcircfunc.asm 内分支指令返回 C 调用函数时设置断点，观察 delays 地址的存储单元，检验大小为 256 的缓冲区初始化时为 0。为了更好地观察屏幕结果，点击右边 Memory 窗口，取消“Float in Main Window”。运行程序，执行到断点后停止，检验是否保存在缓冲区（0x3FE 及 0x3FF）最后（高地址）的最新（16 位）抽样值。存储单元 0x400 中包含最后一个单元地址 0x301，该单元用来存储下一个抽样，该地址是缓冲区的开始。观察核心寄存器并检验 A7 中包含的地址。

再运行程序并观察保存在缓冲区开始的新抽样，注意 A7 增加到了 0x303, 0x305, …更新抽样的方法是很有效的。注意缓冲区大小是以 2 的指数为单位的，这一点很重要。

例 4.15　用外部存储器中的循环缓冲区，用 C 语言调用汇编函数实现 FIR

该例用外部存储器作为循环缓冲区来实现 FIR 滤波器，它对例 4.14 稍微进行了扩展。将例 4.14 中的 C 程序 FIRcirc.c 进行修改，得到 FIRcirc_ext.c（如图 4.36 所示），从而使它调用汇编函数 FIRcircfunc_ext.asm（代替 FIRcircfunc.asm，如图 4.37 所示）。

```
//FIRcirc_ext.c C program calling ASM function using circular buffer

#include "bp1750.cof"                    //BP at 1750 Hz coeff file
int yn = 0;                              //init filter's output

interrupt void c_int11()                 //ISR
{
 short sample_data;
```

```
 sample_data = input_sample();          //newest input sample data
 yn = fircircfunc_ext(sample_data,h,N); //ASM funcn passing to A4,B4,A6
 output_sample(yn >> 15);               //filter's output
 return;                                //return to calling function
}

void main()
{
 comm_intr();                           //init DSK, codec, McBSP
 while(1);                              //infinite loop
}
```

图4.36　使用外部存储器作为循环缓冲区，调用汇编函数的C程序（FIRcirc_ext.c）

```
;FIRcircfunc_ext.asm Function using circular buffer in external memory
;A4=newest sample, B4=coefficient address, A6=filter order
;Delay samples organized: x[n-(N-1)]...x[n]; coeff as h(0)...h[N-1]

            .def    _fircircfunc_ext
            .def    last_addr
            .def    delays
            .sect   "circdata"      ;circular data section
            .align  256             ;align delay buffer 256-byte boundary
delays      .space  256             ;init 256-byte buffer with 0's
last_addr   .int    last_addr-1
            .text                   ;code section
_fircircfunc_ext:                    ;FIR function using circ addr
            MV      A6,A1           ;setup loop count
            MPY     A6,2,A6         ;since dly buffer data as byte
            ZERO    A8              ;init A8 for accumulation

            ADD     A6,B4,B4        ;since coeff buffer data as bytes
            SUB     B4,1,B4         ;B4=bottom coeff array h[N-1]

            MVKL    0x00070040,B6   ;select A7 as pointer and BK0
            MVKH    0x00070040,B6   ;BK0 for 256 bytes (128 shorts)
            MVC     B6,AMR          ;set address mode register AMR

            MVKL    last_addr,A9    ;A9=bottom circ addr in external mem
            MVKH    last_addr,A9    ;(higher 16 bits)in external circ
            LDW     *A9,A7          ;A7=last circ addr
            NOP     4
            STH     A4,*A7++        ;newest sample-->last address
loop:                               ;begin FIR loop
            LDH     *A7++,A2        ;A2=x[n-(N-1)+i] i=0,1,...,N-1
||          LDH     *B4--,B2        ;B2=h[N-1-i] i=0,1,...,N-1
            SUB     A1,1,A1         ;decrement count

  [A1]      B       loop            ;branch to loop if count # 0
            NOP     2
            MPY     A2,B2,A6        ;A6=x[n-(N-1)+i]*h[N-1+i]
```

```
        NOP
        ADD     A6,A8,A8        ;accumulate in A8

        STW     A7,*A9          ;store last circ addr to last_addr
        B       B3              ;return addr to calling routine
        MV      A8,A4           ;result returned in A4
        NOP     4
```

图 4.37 使用外部存储器作为循环缓冲区，C 调用的汇编函数程序（FIRcircfunc_ext.asm）

例中使用的连接命令文件 FIRcirc_ext.cmd 如图 4.38 所示，其中 circdata 段指定的存储区是 buffer_ext，在外部存储器中的起始地址是 0x80000000。

```
/*FIRcirc_ext.cmd Linker file for circular buffer in external memory*/

MEMORY
{
  VECS:        org =          0h,  len =      0x220
  IRAM:        org = 0x00000220,  len = 0x0000FDC0
  buffer_ext:  org = 0x80000000,  len = 0x00000110
  SDRAM:       org = 0x80000110,  len = 0x01000000
  FLASH:       org = 0x90000000,  len = 0x00020000
}
SECTIONS
{
  circdata  :> buffer_ext
  vectors   :> VECS
  .text     :> IRAM
  .bss      :> IRAM
  .cinit    :> IRAM
  .stack    :> IRAM
  .sysmem   :> SDRAM
  .const    :> IRAM
  .switch   :> IRAM
  .far      :> SDRAM
  .cio      :> SDRAM
}
```

图 4.38 使用外部存储器作为循环缓冲区的连接命令文件（FIRcirc_ext.cmd）

建立工程 FIRcirc_ext，观察 delays 地址的存储单元，这样就显示出外部存储区。检验外部存储器中的循环缓冲区，像例 4.14 一样设置断点、仿真、校验存在循环缓冲区末尾的最新抽样以及循环缓冲区开始的下一个抽样。暂停、清除断点，检验输出是中心频率为 1750 Hz 的 FIR 带通滤波器。

参考文献

1. W. J. Gomes III and R. Chassaing, Filter design and implementation using the TMS320C6x interfaced with MATLAB, *Proceedings of the 2000 ASEE Annual Conference*, 2000.

2. A. V. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1989.
3. B. Gold and C. M. Rader, *Digital Signal Processing of Signals*, McGraw-Hill, New York, 1969.
4. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1975.
5. T. W. Parks and J. H. McClellan, Chebychev approximation for nonrecursive digital filter with linear phase, *IEEE Transactions on Circuit Theory*, Vol. CT-19, 1972, pp. 189–194.
6. J. H. McClellan and T. W. Parks, A unified approach to the design of optimum linear phase digital filters, *IEEE Transactions on Circuit Theory*, Vol. CT-20, 1973, pp. 697–701.
7. J. F. Kaiser, Nonrecursive digital filter design using the I0-sinh window function, *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1974.
8. J. F. Kaiser, Some practical considerations in the realization of linear digital filters, *Proceedings of the 3rd Allerton Conference on Circuit System Theory*, Oct. 1965, pp. 621–633.
9. L. B. Jackson, *Digital Filters and Signal Processing*, Kluwer Academic, Norwell, MA, 1996.
10. J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, Prentice Hall, Upper Saddle River, NJ, 1996.
11. R. G. Lyons, *Understanding Digital Signal Processing*, Addison-Wesley, Reading, MA, 1997.
12. F. J. Harris, On the use of windows for harmonic analysis with the discrete Fourier transform, *Proceedings of the IEEE*, Vol. 66, 1978, pp. 51–83.
13. I. F. Progri, W. R. Michalson, and R. Chassaing, Fast and efficient filter design and implementation on the TMS320C6711 digital signal processor, *2001 International Conference on Acoustics, Speech, and Signal Processing Student Forum*, May 2001.
14. B. Porat, *A Course in Digital Signal Processing*, Wiley, New York, 1997.
15. T. W. Parks and C. S. Burrus, *Digital Filter Design*, Wiley, New York, 1987.
16. S. D. Stearns and R. A. David, *Signal Processing in Fortran and C*, Prentice Hall, Upper Saddle River, NJ, 1993.
17. N. Ahmed and T. Natarajan, *Discrete-Time Signals and Systems*, Reston Publishing, Reston, VA, 1983.
18. S. J. Orfanidis, *Introduction to Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1996.
19. A. Antoniou, *Digital Filters: Analysis, Design, and Applications*, McGraw-Hill, New York, 1993.
20. E. C. Ifeachor and B. W. Jervis, *Digital Signal Processing: A Practical Approach*, Addison-Wesley, Reading, MA, 1993.
21. P. A. Lynn and W. Fuerst, *Introductory Digital Signal Processing with Computer Applications*, Wiley, New York, 1994.
22. R. D. Strum and D. E. Kirk, *First Principles of Discrete Systems and Digital Signal Processing*, Addison-Wesley, Reading, MA, 1988.
23. D. J. DeFatta, J. G. Lucas, and W. S. Hodgkiss, *Digital Signal Processing: A System Approach*, Wiley, New York, 1988.

24. C. S. Williams, *Designing Digital Filters*, Prentice Hall, Upper Saddle River, NJ, 1986.
25. R. W. Hamming, *Digital Filters*, Prentice Hall, Upper Saddle River, NJ, 1983.
26. S. K. Mitra and J. F. Kaiser, eds., *Handbook for Digital Signal Processing*, Wiley, New York, 1993.
27. S. K. Mitra, *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998.
28. R. Chassaing, B. Bitler, and D. W. Horning, Real-time digital filters in C, *Proceedings of the 1991 ASEE Annual Conference*, June 1991.
29. R. Chassaing and P. Martin, Digital filtering with the floating-point TMS320C30 digital signal processor, *Proceedings of the 21st Annual Pittsburgh Conference on Modeling and Simulation*, May 1990.
30. S. D. Stearns and R. A. David, *Signal Processing in Fortran and C*, Prentice Hall, Upper Saddle River, NJ, 1993.
31. R. A. Roberts and C. T. Mullis, *Digital Signal Processing*, Addison-Wesley, Reading, MA, 1987.
32. E. P. Cunningham, *Digital Filtering: An Introduction*, Houghton Mifflin, Boston, 1992.
33. N. J. Loy, *An Engineer's Guide to FIR Digital Filters*, Prentice Hall, Upper Saddle River, NJ, 1988.
34. H. Nuttall, Some windows with very good sidelobe behavior, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-29, No. 1, Feb. 1981.
35. L. C. Ludemen, *Fundamentals of Digital Signal Processing*, Harper & Row, New York, 1986.
36. M. Bellanger, *Digital Processing of Signals: Theory and Practice*, Wiley, New York, 1989.
37. M. G. Bellanger, *Digital Filters and Signal Analysis*, Prentice Hall, Upper Saddle River, NJ, 1986.
38. F. J. Taylor, *Principles of Signals and Systems*, McGraw-Hill, New York, 1994.
39. F. J. Taylor, *Digital Filter Design Handbook*, Marcel Dekker, New York, 1983.
40. W. D. Stanley, G. R. Dougherty, and R. Dougherty, *Digital Signal Processing*, Reston Publishing, Reston, VA, 1984.
41. R. Kuc, *Introduction to Digital Signal Processing*, McGraw-Hill, New York, 1988.
42. H. Baher, *Analog and Digital Signal Processing*, Wiley, New York, 1990.
43. J. R. Johnson, *Introduction to Digital Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1989.
44. S. Haykin, *Modern Filters*, Macmillan, New York, 1989.
45. T. Young, *Linear Systems and Digital Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1985.
46. A. Ambardar, *Analog and Digital Signal Processing*, PWS, MA, 1995.
47. A. W. M. van den Enden and N. A. M. Verhoeckx, *Discrete-Time Signal Processing*, Prentice-Hall International, Hemel Hempstead, Hertfordshire, England, 1989.
48. MATLAB, MathWorks, Natick, MA.
49. R. Chassaing and D. W. Horning, *Digital Signal Processing with the TMS320C25*, Wiley, New York, 1990.

第 5 章　无限冲激响应滤波器

本章介绍的主要内容包括：（1）无限冲激响应滤波器的结构（直接 I 型、直接 II 型、串联型和并联型）；（2）滤波器的双线性变换设计方法；（3）利用差分方程产生正弦波的方法；（4）滤波器的设计方法和实用程序包；（5）TMS320C6x 和 C 语言程序实例。

第 4 章讨论的 FIR 滤波器没有对应的模拟部分作为参照，本章我们讨论利用已有的模拟滤波器知识设计无限冲激响应（IIR）滤波器。设计过程涉及到用双线性变化法（BLT）将模拟滤波器变换到数字滤波器。同样地，双线性变换法能将 s 域模拟滤波器传输函数变换到等效的 z 域数字滤波器的传输函数。

5.1　引言

考察下面的输入输出方程：

$$y(n)=\sum_{k=0}^{N}a_k x(n-k)-\sum_{j=1}^{M}b_j y(n-j) \tag{5.1}$$

$$\begin{aligned}&=a_0x(n)+a_1x(n-1)+a_2x(n-2)+\cdots+a_Nx(n-N)\\&\quad -b_1y(n-1)-b_2y(n-2)-\cdots-b_My(n-M)\end{aligned} \tag{5.2}$$

这种递归形式的方程表示 IIR 滤波器，输出不仅与现在的输入有关，还与过去的输出有关（有反馈）。即在 n 时刻，输出 $y(n)$ 不仅与当前的输入 $x(n)$ 有关，还与过去的输入 $x(n-1)$，$x(n-2)$，…，$x(n-N)$ 和过去的输出 $y(n-1)$，$y(n-2)$，…，$y(n-M)$ 有关。

如果假设式（5.2）的所有初始条件全为 0，则式（5.2）的 z 变换为：

$$\begin{aligned}Y(z)&=a_0X(z)+a_1z^{-1}X(z)+a_2z^{-2}X(z)+\cdots+a_Nz^{-N}X(z)\\&\quad -b_1z^{-1}Y(z)-b_2z^{-2}Y(z)-\cdots-b_Mz^{-M}Y(z)\end{aligned} \tag{5.3}$$

令式（5.3）中的 $N=M$，那么传输函数 $H(z)$ 可写为：

$$H(z)=\frac{Y(z)}{X(z)}=\frac{a_0+a_1z^{-1}+a_2z^{-2}+\cdots+a_Nz^{-N}}{1+b_1z^{-1}+b_2z^{-2}+\cdots+b_Nz^{-N}}=\frac{N(z)}{D(z)} \tag{5.4}$$

这里 $N(z)$ 与 $D(z)$ 分别表示分子分母多项式。将分子分母同时乘以 z^N，$H(z)$ 变为：

$$H(z)=\frac{a_0z^N+a_1z^{N-1}+a_2z^{N-2}+\cdots+a_N}{z^N+b_1z^{N-1}+b_2z^{N-2}+\cdots+b_N}=C\prod_{i=1}^{N}\frac{z-z_i}{z-p_i} \tag{5.5}$$

这是有 N 个零点和 N 个极点的传输函数。如果所有的系数 b_j 均为 0，那么传输函数就简化成第 4 章中讨论过的在 z 平面原点有 N 个极点的 FIR 滤波器的传输函数。由第 4 章的讨论可知，为了使系统稳定，所有极点必须在单位圆内。因此为使 IIR 滤波器稳定，每个极点的模必须小于 1，也就是：

1. 如果 $|P_i|<1$，那么当 $n\to\infty$，$h(n)\to 0$ 时，系统是稳定系统。
2. 如果 $|P_i|>1$，那么当 $n\to\infty$，$h(n)\to\infty$ 时，系统是不稳定系统。

如果$|P_i| = 1$，则系统就是产生振荡响应的临界稳定系统。另外，单位圆上有多阶极点的系统也是不稳定的。还需要注意的是，如果所有系数$b_j = 0$，则系统就变成了非递归的稳定 FIR 滤波器。

5.2　IIR 滤波器的结构

下面将讨论几种 IIR 滤波器的结构。

5.2.1　直接 I 型结构

直接 I 型结构如图 5.1 所示，式（5.2）描述的滤波器可用这种结构实现。图 5.1 隐含有求和的功能，图中没有显示出来。对一个 N 阶滤波器，这种结构需要 $2N$ 个延时单元，用z^{-1}表示。例如，二阶滤波器的阶数$N = 2$，因此需要 4 个延时单元。

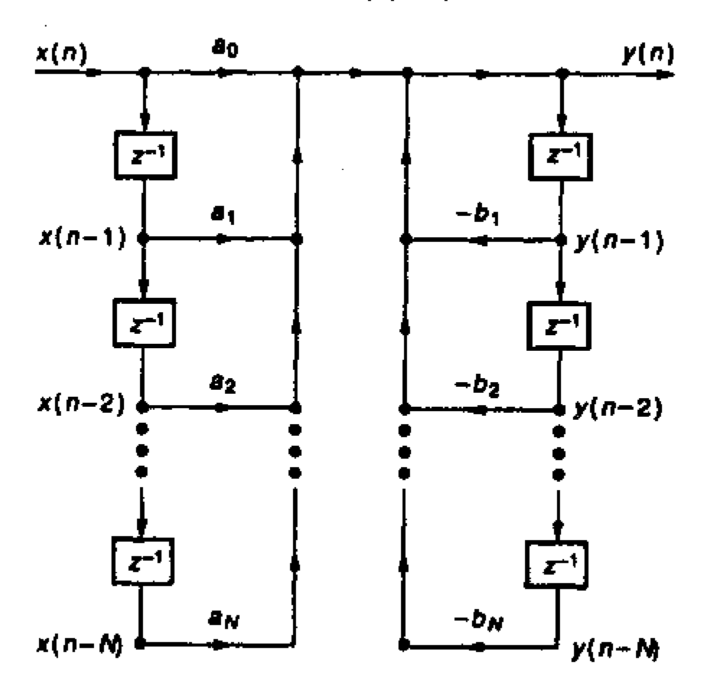


图 5.1　直接 I 型 IIR 滤波器的结构

5.2.2　直接 II 型结构

直接 II 型滤波器的结构如图 5.2 所示，这是最常用的滤波器结构之一，这种结构需要的延时单元是直接 I 型结构的一半。例如，直接 II 型结构的二阶滤波器需要两个延时单元，而直接 I 型结构需要 4 个延时单元。为了能用直接 II 型结构实现式（5.2），定义延迟变量$U(z)$：

$$U(z) = \frac{X(z)}{D(z)} \tag{5.6}$$

其中$D(z)$是式（5.4）传输函数的分母多项式。由式（5.4）和式（5.6），$Y(z)$可写为：

$$\begin{aligned} Y(z) &= \frac{N(z)X(z)}{D(z)} = N(z)U(z) \\ &= U(z)(a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}) \end{aligned} \tag{5.7}$$

这里$N(z)$是式（5.4）传输函数的分子多项式，由式（5.6）可知：

$$X(z)=U(z)D(z)=U(z)(1+b_1z^{-1}+b_2z^{-2}+\cdots+b_Nz^{-N}) \tag{5.8}$$

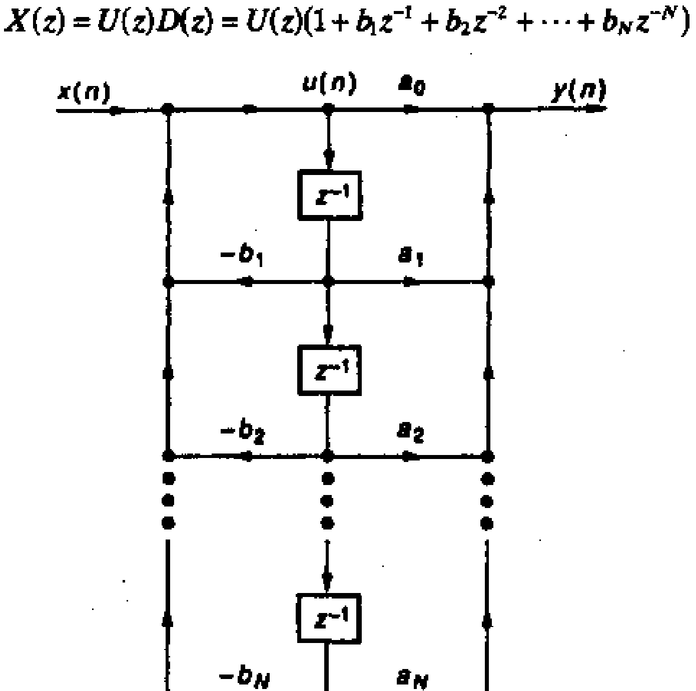


图 5.2　直接Ⅱ型IIR滤波器的结构

对式（5.8）做z的逆变换：

$$x(n)=u(n)+b_1u(n-1)+b_2u(n-2)+\cdots+b_Nu(n-N) \tag{5.9}$$

对式（5.9）的$u(n)$求解，得到：

$$u(n)=x(n)-b_1u(n-1)-b_2u(n-2)-\cdots-b_Nu(n-N) \tag{5.10}$$

对式（5.7）做z的逆变换，可得：

$$y(n)=a_0u(n)+a_1u(n-1)+a_2u(n-2)+\cdots+a_Nu(n-N) \tag{5.11}$$

直接Ⅱ型结构可用式（5.10）和式（5.11）表示，图5.2中的延时单元满足式（5.10），图5.2中的输出$y(n)$满足式（5.11）。

式（5.10）与式（5.11）用于编程实现IIR滤波器，$u(n-1)$，$u(n-2)$，…的初始值设为0。在时刻n，得到新的抽样$x(n)$，用式（5.10）求出$u(n)$，这时滤波器的输出变为：

$$y(n)=a_0u(n)+0$$

在$n+1$时刻，得到新抽样$x(n+1)$，式（5.10）中的延时变量被更新，也就是：

$$u(n+1)=x(n+1)-b_1u(n)-0$$

这里$u(n-1)$被$u(n)$替代。由式（5.11），$n+1$时刻的输出为：

$$y(n+1)=a_0u(n+1)+a_1u(n)+0$$

同样，$n+2$，$n+3$，…时刻的输出也可计算出来。对每个特定时刻，当采集到每个新的输入抽样时，延时变量以及输出就可以根据式（5.10）和式（5.11）分别计算出来。

5.2.3　直接 II 型的转置

直接 II 型转置结构是直接 II 型结构的改进形式，它需要相同数目的延时单元。由直接 II 型结构变成直接 II 型转置结构需要以下步骤：

1. 将所有分支的方向变成相反方向；
2. 将输入与输出倒过来（原输入变成输出，原输出变成输入）；
3. 重画结构图（和通常的结构一样），把输入节点放在左边，输出节点放在右边。

直接 II 型的转置结构如图 5.3 所示。为了证明这一点，设$u_0(n)$和$u_1(n)$如图 5.3 所示，由转置结构可得：

$$u_0(n) = a_2 x(n) - b_2 y(n) \qquad (5.12)$$

$$u_1(n) = a_1 x(n) - b_1 y(n) + u_0(n-1) \qquad (5.13)$$

$$y(n) = a_0 x(n) + u_1(n-1) \qquad (5.14)$$

用式（5.12）求出$u_0(n-1)$，代入式（5.13）可得：

$$u_1(n) = a_1 x(n) - b_1 y(n) + [a_2 x(n-1) - b_2 y(n-1)] \qquad (5.15)$$

用式（5.15）求出$u_1(n-1)$，代入式（5.14），得到：

$$y(n) = a_0 x(n) + [a_1 x(n-1) - b_1 y(n-1) + a_2 x(n-2) - b_2 y(n-2)] \qquad (5.16)$$

该式和式（5.2）一样，是二阶系统的通用输入输出方程。这种转置结构先实现零点，然后再实现极点，而直接 II 型结构先实现极点。

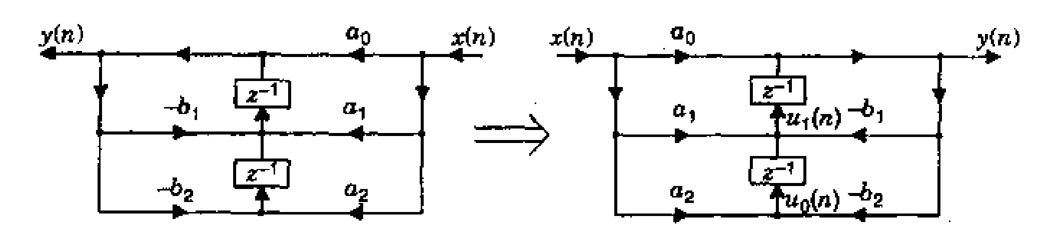


图 5.3　直接 II 型转置 IIR 滤波器结构

5.2.4　串联结构

式（5.5）的传输函数可分解成一阶或二阶传输函数积的形式，如：

$$H(z) = CH_1(z)H_2(z)\cdots H_r(z) \qquad (5.17)$$

级联（或串联）结构如图 5.4 所示，整个传输函数可表示成多个传输函数级联的形式。对每个级联单元，可用直接 II 型和它的转置结构表示。图 5.5 给出了一个用两个直接 II 型二阶单元级联实现四阶 IIR 滤波器的级联结构。传输函数$H(z)$用二阶传输函数级联形式表示，可写为：

$$H(z) = \prod_{i=1}^{N/2} \frac{a_{0i} + a_{1i}z^{-1} + a_{2i}z^{-2}}{1 + b_{1i}z^{-1} + b_{2i}z^{-2}} \qquad (5.18)$$

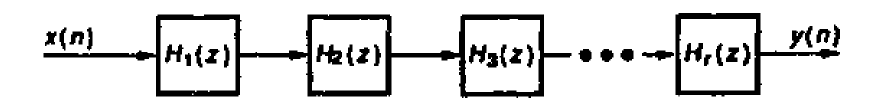


图 5.4 级联型 IIR 滤波器结构

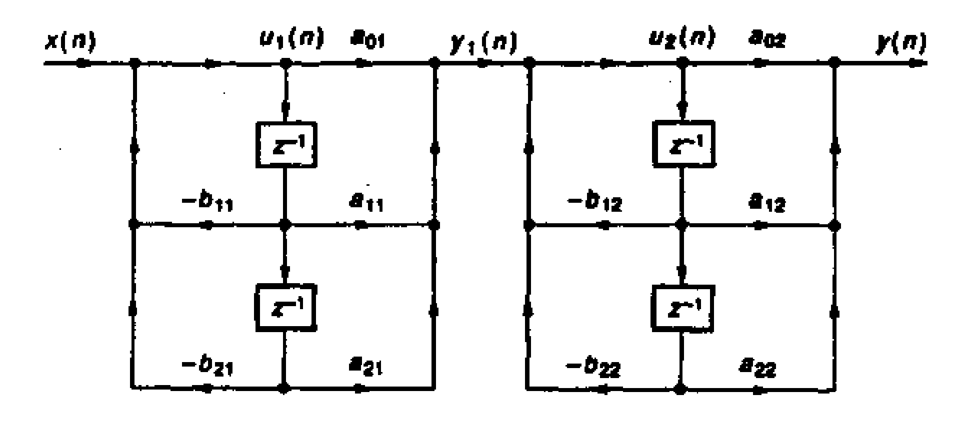


图 5.5 两个二阶直接 II 型单元级联实现四阶滤波器

式（5.17）中的常数 C 可合并到系数中去，每个部分用下标 i 表示。例如，对$N=4$的四阶传输函数，$H(z)$可写为：

$$H(z)=\frac{(a_{01}+a_{11}z^{-1}+a_{21}z^{-2})(a_{02}+a_{12}z^{-1}+a_{22}z^{-2})}{(1+b_{11}z^{-1}+b_{21}z^{-2})(1+b_{12}z^{-1}+b_{22}z^{-2})} \tag{5.19}$$

式（5.19）可用图 5.5 来检验。从数学的角度来看，适当改变分子分母多项式的因子位置不会影响输出；但从实际实现的角度来看，适当改变二阶单元的位置可使量化噪声[1-5]最小。注意：这里第一级单元的输出$y_1(n)$变成第二级的输入，中间输出结果保存到寄存器中，对它提前截短所造成的影响可以忽略。我们将举例说明用二阶直接 II 型级联形式实现 IIR 滤波器的方法。

5.2.5 并联结构

式（5.5）的传输函数可表示为：

$$H(z)=C+H_1(z)+H_2(z)+\cdots+H_r(z) \tag{5.20}$$

该式可用部分分式法（PEE）得到，图 5.6 给出了并联结构传输函数，每个部分传输函数$H_1(z)$，$H_2(z)$，…可以是一阶或二阶函数。就像级联型结构一样，并联型结构可以有效地用二阶直接 II 型结构表示出来。$H(z)$可表示为：

$$H(z)=C+\sum_{i=1}^{N/2}\frac{a_{0i}+a_{1i}z^{-1}+a_{2i}z^{-2}}{1+b_{1i}z^{-1}+b_{2i}z^{-2}} \tag{5.21}$$

例如对于一个四阶传输函数，式（5.21）中的$H(z)$可表示为：

$$H(z)=C+\frac{a_{01}+a_{11}z^{-1}+a_{21}z^{-2}}{1+b_{11}z^{-1}+b_{21}z^{-2}}+\frac{a_{02}+a_{12}z^{-1}+a_{22}z^{-2}}{1+b_{12}z^{-1}+b_{22}z^{-2}} \tag{5.22}$$

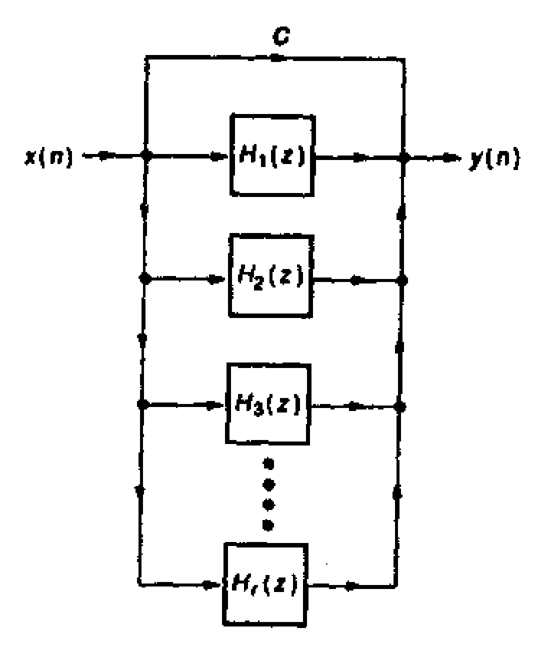


图 5.6　并联型 IIR 滤波器结构

图 5.7 用两个直接 II 型结构单元表示成一个四阶并联型结构的 IIR 滤波器。由图 5.7，输出 $y(n)$ 可表示成每个并联单元的输出之和的形式，也就是：

$$y(n)=Cx(n)+\sum_{i=1}^{N/2}y_i(n) \tag{5.23}$$

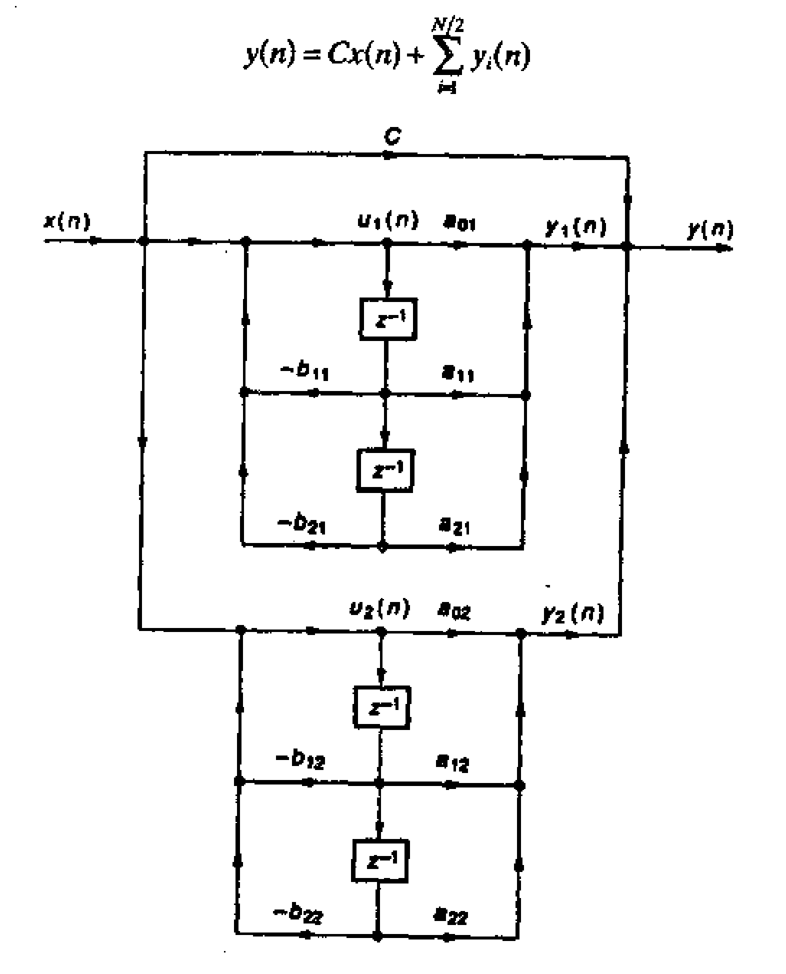


图 5.7　两个直接 II 型结构单元并联组成的四阶 IIR 滤波器

还有其他形式的滤波器结构，像格型结构，这种滤波器结构在语音和自适应滤波处理中非常有用。虽然这种结构的计算效率不如直接 II 型或级联型结构，需要较多的乘法运算，但量化效应对它影响较小[6-8]。IIR 滤波器系数的量化误差取决于复平面上零极点的位置变化情况，这就意味着一个特定极点位置的偏移依赖于其他位置的所有极点。为减小这种极点间的相互依赖性，通常用二阶单元级联实现 N 阶滤波器。

5.3　双线性变换法

双线性变换法（BLT）是把模拟滤波器转变成数字滤波器最常用的方法，它使用式（5.24）实现 s 平面到 z 平面的一一映射：

$$s = K\frac{z-1}{z+1} \tag{5.24}$$

其中的常数 K 通常选择为 $K = 2/T$，T 表示抽样变量，K 也可以选择其他值，因为在设计过程中它没有影响。为了方便说明双线性变换过程，我们取 $K = 1$ 或 $T = 2$，代入式（5.24），可得：

$$z = \frac{1+s}{1-s} \tag{5.25}$$

这种变换实现了下面的映射：

1. 将 s 平面的左半平面，对应 $\sigma < 0$ 的区域，映射到 z 平面单位圆内；
2. 将 s 平面的右半平面，对应 $\sigma > 0$ 的区域，映射到 z 平面单位圆外；
3. 将 s 平面的虚轴，映射到 z 平面的单位圆上。

令 ω_A，ω_D 分别表示模拟频率和数字频率，$s = j\omega_A$，$z = e^{j\omega_D T}$，代入式（5.24）得：

$$j\omega_A = \frac{e^{j\omega_D T}-1}{e^{j\omega_D T}+1} = \frac{e^{j\omega_D T/2}\left(e^{j\omega_D T/2}-e^{-j\omega_D T/2}\right)}{e^{j\omega_D T/2}\left(e^{j\omega_D T/2}+e^{-j\omega_D T/2}\right)} \tag{5.26}$$

根据用复指数函数表示正弦和余弦的欧拉公式，式（5.26）变为：

$$\omega_A = \tan\frac{\omega_D T}{2} \tag{5.27}$$

这样就把模拟频率和数字频率联系起来。图 5.8 给出了 ω_A 为正数时两者的关系。ω_A 对应于 0 到 1 区间映射为 ω_D 的 0 到 $\omega_s/4$ 比较好的线性区间，这里 ω_s 是以弧度为单位的抽样频率。但 $\omega_A > 1$ 的整个区间却是极其非线性的，并映射为 ω_D 的 $\omega_s/4$ 到 $\omega_s/2$ 区间，在这个区域内的压缩现象称为频率扭曲现象，因此，需要用预畸的方法来弥补这种失真，频率 ω_A 和 ω_D 应满足：

$$H(s)\big|_{s=j\omega_A} = H(z)\big|_{z=e^{j\omega_D T}} \tag{5.28}$$

5.3.1　双线性变换法设计过程

双线性变换法利用已知的模拟滤波器传输函数来设计数字滤波器，它能使用成熟的模拟滤波器传输函数（巴特沃斯、切比雪夫等）来设计数字滤波器，附录 D 介绍了用 MATLAB 实现的几种形式的滤波器。切比雪夫 I 型和 II 型滤波器分别在通带和阻带内实现了等波动响应。对于一个给定指标，这些滤波器比通带和阻带都是单调变化的巴特沃斯滤波器的阶数要更低。椭圆滤波器在通带和阻带都是等波动的，它的阶数比切比雪夫型滤波器更低，但它更难设计，因为它的相位响应在通带内是非线性的。虽然巴特沃斯滤波器需要更高的阶数，它的相位响应在通带内是线性的。

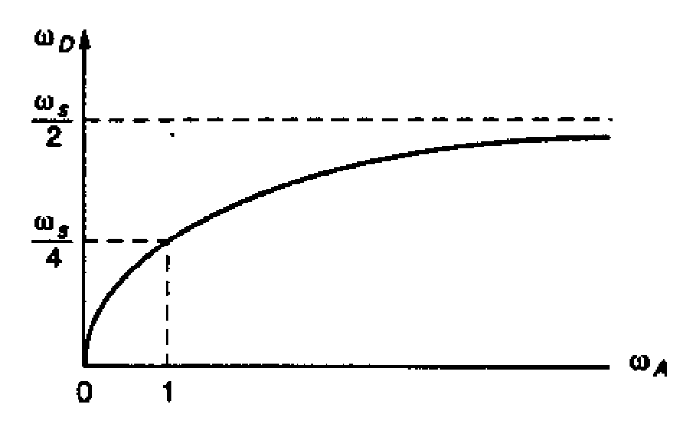


图 5.8 模拟频率和数字频率间的关系

为了利用双线性变换法求出$H(z)$，需要以下几个步骤：

1. 找一个已知的模拟滤波器传输函数$H(s)$;
2. 预畸要求的数字频率ω_D，得到式（5.27）的模拟频率ω_A;
3. 对选择的模拟滤波器传输函数频率除以ω_A，代入式（5.29）：

$$H(s)|_{s=s/\omega_A} \tag{5.29}$$

4. 用双线性式（5.24）求得到$H(z)$，也就是：

$$H(z)=H(s/\omega_A)|_{s=(z-1)/(z+1)} \tag{5.30}$$

对于带通和带阻滤波器，它们分别有较低和较高截止频率ω_{D1}和ω_{D2}，这时需要求两个模拟频率ω_{A1}和ω_{A2}，附录 E 中的练习进一步说明了双线性变换过程。

5.4 设计 IIR 的 C 语言程序实例

本节介绍 5 个程序例子，说明如何用级联直接 II 型结构单元实现 IIR 滤波器以及用差分方程产生音频信号。

例 5.1 用二阶单元的级联结构实现 IIR 滤波器

图 5.9 给出了一个用二阶单元级联实现通用 IIR 滤波器的 C 程序，该程序每个单元使用了下面的两个方程：

$$u(n) = x(n) - b_1u(n-1) - b_2u(n-2)$$
$$y(n) = a_0u(n) + a_1u(n-1) + a_2u(n-2)$$

对每个 n 或抽样间隔，程序的循环部分执行了 5 次（级联单元的级数）。对于第一个级联单元，$x(n)$是一个新采集的输入抽样，但对其他级联单元，输入 $x(n)$就是前一级的输出 $y(n)$。

系数 b[*i*][0]和 b[*i*][1]分别对应 b_1 和 b_2，i 表示属于哪一级，延时 dly[*i*][0]和 dly[*i*][1]分别对应 $u(n-1)$和 $u(n-2)$。

```
//IIR.c IIR filter using cascaded Direct Form II
//Coefficients a's and b's correspond to b's and a's, from MATLAB

#include "bs1750.cof"                    //BS @ 1750Hz coefficient file
short dly[stages][2] = {0};              //delay samples per stage

interrupt void c_int11()                 //ISR
 {
 int i, input;
 int un, yn;

 input = input_sample();                 //input to 1st stage
 for (i = 0; i < stages; i++)            //repeat for each stage
  {
   un=input-((b[i][0]*dly[i][0])>>15) - ((b[i][1]*dly[i][1])>>15);

   yn=((a[i][0]*un)>>15)+((a[i][1]*dly[i][0])>>15)+((a[i][2]*dly[i][1])>>15);

   dly[i][1] = dly[i][0];                //update delays
   dly[i][0] = un;                       //update delays
   input = yn;                           //intermediate output->in to next stage
  }
  output_sample(yn);                     //output final result for time n
  return;                                //return from ISR
}

void main()
{
  comm_intr();                           //init DSK, codec, McBSP
  while(1);                              //infinite loop
}
```

图 5.9　二阶单元级联实现 IIR 的 C 程序（IIR.c）

IIR 带阻滤波器

系数文件 bs1750.cof（如图 5.10 所示）是由附录 D 得到的，它表示一个 10 阶带阻滤波器，是用 MATLAB 的 SPTOOL 工具设计的，如图 D.2 所示。注意 MATLAB 设计工具显示阶数是 5，是指使用的二阶级联单元的数目。系数文件包括分子系数（每级三个）a's 和分母系数 b's（每级两个）。这本书里用的 a's 和 b's 分别对应 MATLAB 中的 b's 和 a's。

建立并运行工程 IIR，检验输出是否是中心频率为 1750 Hz 的 IIR 带阻滤波器。图 5.11 是利用 HP 分析仪得到带阻滤波器的输出频率响应的结果（这里用噪声作为分析仪输入）。

IIR 带通和低通滤波器

1. 用系数文件 bp2000.cof（在辅助材料中）重新建立该工程，它表示一个 36 阶（18 级）切比雪夫 II 型 IIR 带通滤波器，中心频率为 2 kHz。滤波器是用 MATLAB 设计的，如图 5.12 所示，检验滤波器的输出是中心频率为 2 kHz 的 IIR 带通滤波器。图 5.13 是该 36 阶 IIR 带通滤波器的输出频率响应，它是用 HP 分析仪获得的。
2. 用系数文件 lp2000.cof（在辅助材料中）重新建立该工程，它表示一个 8 阶 IIR 低通滤波器，截止频率为 2 kHz（同样是用 MATLAB 来设计的），检验低通 IIR 滤波器的输出是否正确。

```
//bs1750.cof IIR bandstop coefficient file, centered at 1,750Hz

#define stages 5                      //number of 2nd-order stages

int a[stages][3]=          {          //numerator coefficients
{27940, -10910, 27940},               //a10, a11, a12 for 1st stage
{32768, -11841, 32768},               //a20, a21, a22 for 2nd stage
{32768, -13744, 32768},               //a30, a31, a32 for 3rd stage
{32768, -11338, 32768},               //a40, a41, a42 for 4th stage
{32768, -14239, 32768}  };

int b[stages][2]=          {          //*denominator coefficients
{-11417, 25710},                      //b11, b12 for 1st stage
{-9204, 31581},                       //b21, b22 for 2nd stage
{-15860, 31605},                      //b31, b32 for 3rd stage
{-10221, 32581},                      //b41, b42 for 4th stage
{-15258, 32584}            };         //b51, b52 for 5th stage
```

图 5.10　MATLAB 设计的 10 阶 IIR 带阻滤波器系数文件（bs1750.cof），参见附录 D

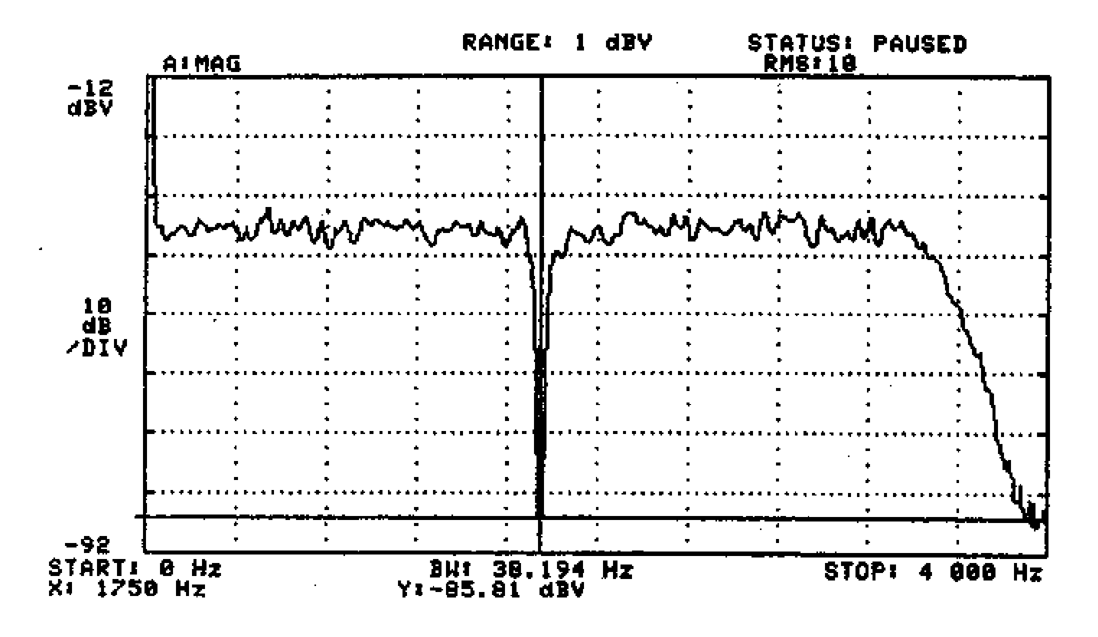


图 5.11　HP 分析仪测得的 10 阶 IIR，滤波器中心频率为 1750 Hz

例 5.2　用两个二阶差分方程产生两个音频音

本例使用差分方程产生两个音频音并将它们相加，输出保存在存储器中，并用 CCS 画出相应的图形。产生正弦波的差分方程是：

$$y(n) = Ay(n-1) - y(n-2)$$

其中

$$A = 2\cos(\omega T)$$
$$y(-1) = -\sin(\omega T)$$
$$y(-2) = -\sin(2\omega T)$$

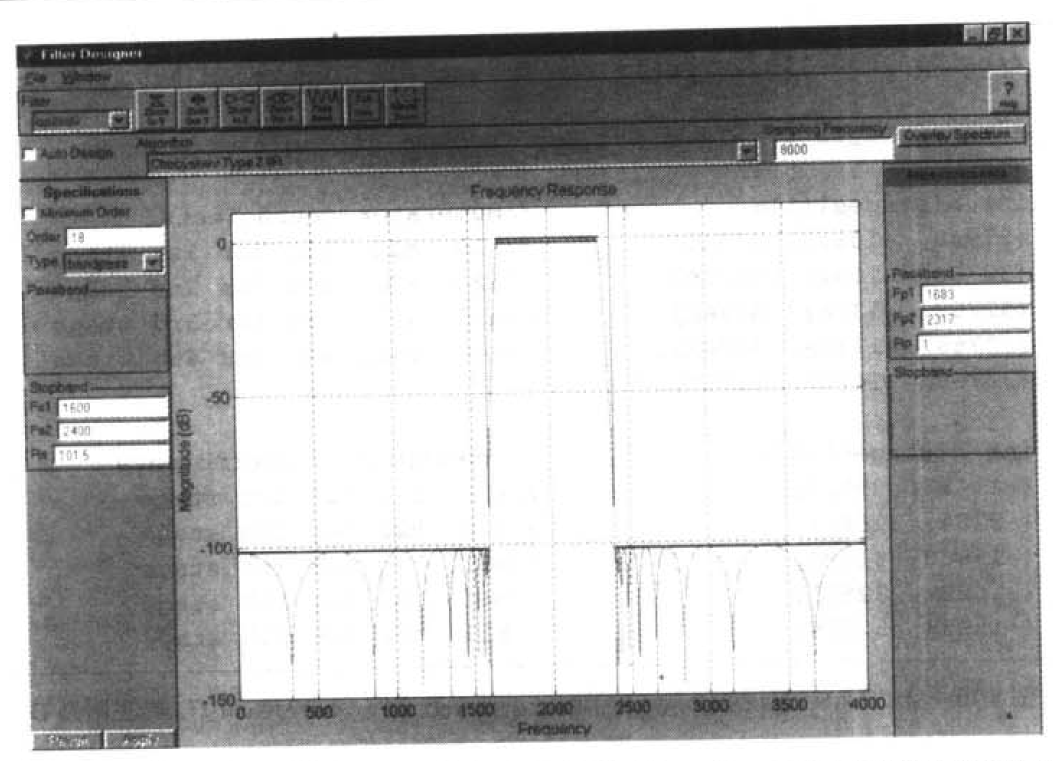


图 5.12 MATLAB 滤波器设计工具（SPTOOL）显示的 36 阶 IIR 带通滤波器的频率特性

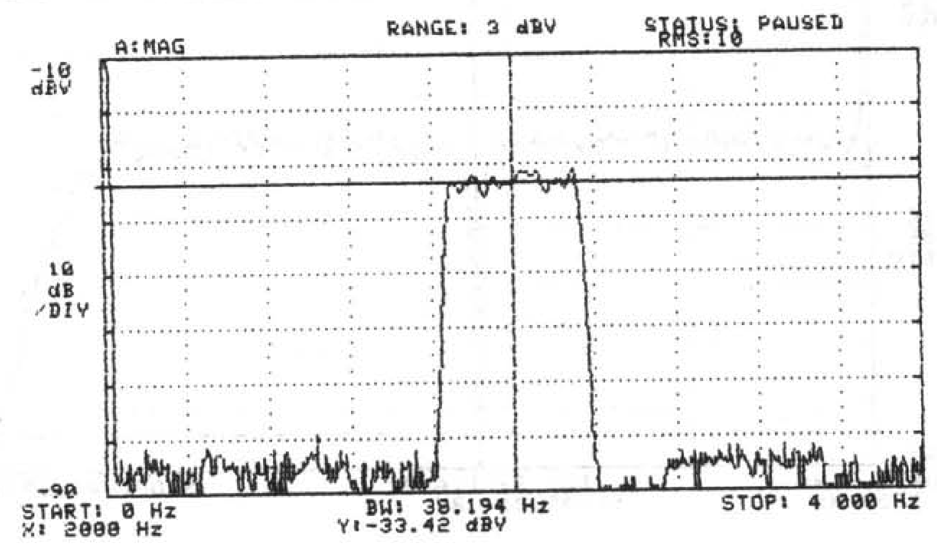


图 5.13 HP 分析仪得到的 36 阶 IIR 带通滤波器的输出频率响应，滤波器中心频率为 2000 Hz

$y(-1)$和 $y(-2)$是两个初始条件，$\omega = 2\pi f$，抽样周期 $T = 1/F_s = 1/(8\ \text{kHz}) = 0.125\ \text{ms}$。$y(n)$的z变换为：

$$Y(z) = A\{z^{-1}Y(z) + y(-1)\} - \{z^{-2}Y(z) + z^{-1}y(-1) + y(-2)\}$$

也可写成：

$$\begin{aligned} Y(z)\{1 - Az^{-1} + z^{-2}\} &= Ay(-1) - z^{-1}\,y(-1) - y(-2) \\ &= -2\cos(\omega T)\sin(\omega T) + z^{-1}\sin(\omega T) + \sin(2\omega T) \\ &= z^{-1}\,\sin(\omega T) \end{aligned}$$

求得$Y(z)$为：

$$Y(z) = z\sin(\omega T)/(z^2 - Az + 1)$$

$Y(z)$ 的z的逆变换是：

$$y(n) = ZT^{-1}\{Y(z)\} = \sin(n\omega T)$$

当$f = 1.5$ kHz 时：

$$\begin{aligned} A = 2\cos(\omega T) &= 0.765 \rightarrow A\times 2^{14} = 12\,540 \\ y(-1) = -\sin(\omega T) &= -0.924 \rightarrow y(-1)\times 2^{14} = -15\,137 \\ y(-2) = -\sin(2\omega T) &= -0.707 \rightarrow y(-2)\times 2^{14} = -11\,585 \end{aligned}$$

当$f = 2$ kHz 时：

$$\begin{aligned} &A = 0 \\ &y(-1) = -1 \rightarrow y(-1)\times 2^{14} = -16\,384 \\ &y(-2) = 0 \end{aligned}$$

二阶差分方程的系数A和两个初始条件决定产生的频率。为了用定点实现，需要对它们进行定标（归一化处理）。使用差分方程：

$$y(n) = Ay(n-1) - y(n-2)$$

$n = 0$时的输出为：

$$y(0) = Ay(-1) - y(-2) = -2\cos(\omega T)\sin(\omega T) + \sin(2\omega T) = 0$$

图 5.14 给出了用差分方程产生音频音的 C 程序 two_tones.c，数组 y1[3]保存生成 1.5 kHz 音频音的三个数据：$y1(0)$，$y1(-1)$和$y1(-2)$；数组 y2[3]保存生成 2 kHz 音频音的三个数据：$y2(0)$，$y2(-1)$和$y2(-2)$。函数 sinegen 使用二阶差分方程产生每个音频音，再把两个信号相加。为了定点实现得到更好的结果，将每个数同时除以2^{14}。

建立并运行工程 two_tones，检验输出是 1.5 kHz 和 2 kHz 两个音频音的和，输出也保存在存储缓冲区中。使用 CCS 画出如图 5.15 所示的两个正弦信号的 FFT 幅度谱。缓冲区的起始地址是 sinegen_buffer（也可参见例 1.2）。

上述技术也可用于产生双音多频信号，例如，产生 697 Hz 和 1209 Hz 的两个音频信号并把它们相加，这正好对应电话中的按键“3”。

例 5.3　用差分方程产生正弦波形

该例用另一个差分方程也同样产生正弦音频音，可参见产生两个音频音并对它们进行相加的例 5.2。考察第 4 章中得到的二阶差分方程：

$$y(n) = Ay(n-1) + By(n-2) + Cx(n-1)$$

这里$B = -1$。当$n = 1$时，使用一个冲激信号，这样得到$x(n-1) = x(0) = 1$，其他均为 0。对于$n = 1$，有：

$$y(1) = Ay(0) + By(-1) + Cx(0) = C$$

其中$y(0) = 0$，$y(-1) = 0$。对于$n \geqslant 2$，有：

$$y(n) = Ay(n-1) - y(n-2)$$

给定抽样时间$T = 1/F_s$和所需频率ω，计算系数$A = 2\cos(\omega T)$，$C = \sin(\omega T)$。

当$f = 1.5$ kHz 时：

$$A = 2\cos(\omega T) = 0.765 \rightarrow A\times 2^{14} = 12\,540$$
$$y(1) = C = 0.924 \rightarrow C\times 2^{14} = 15\,137$$
$$y(2) = Ay(1) = 0.707 \rightarrow y(2)\times 2^{14} = 11\,585$$

当 $f = 2$ kHz 时：

$$A = 2\cos(\omega T) = 0$$
$$y(1) = C = \sin(\omega T) = 1 \rightarrow C\times 2^{14} = 16\,384$$
$$y(2) = Ay(1) - y(0) = AC = 0$$

```
//two_tones.c Generates/adds two tones using difference equations

short sinegen(void);                   //for generating tone
short output;                          //for output
short sinegen_buffer[256];             //buffer for output data
const short bufferlength = 256;        //buffer size for plot with CCS
short i = 0;                           //buffer count index

short y1[3] = {0,-15137,-11585};       //y1(0),y1(-1),y1(-2) for 1.5kHz
const short A1 = 12540;                //A1 = 2coswT scaled by 2^14
short y2[3] = {0,-16384,0};            //y2(0),y2(-1),y2(-2) for 2kHz
const short A2 = 0;                    //A2 = 2coswT scaled by 2^14

interrupt void c_int11()               //ISR
{
 output = sinegen();                   //out from tone generation function
 sinegen_buffer[i] = output;           //output into buffer
 output_sample(output);                //output result
 i++;                                  //increment buffer count
 if (i == bufferlength) i = 0;         //if buffer count=size of buffer
 return;                               //return to main
}

short sinegen()                                      //function to generate tone
{
 y1[0] =((((int)y1[1]*(int)A1))>>14)-y1[2];         //y1(n)=A1*y1(n-1)-y1(n-2)
 y1[2] = y1[1];                                      //update y1(n-2)
 y1[1] = y1[0];                                      //update y1(n-1)

 y2[0] =((((int)y2[1]*(int)A2))>>14)-y2[2];         //y2(n)=A2*y2(n-1)-y2(n-2)
 y2[2] = y2[1];                                      //update y2(n-2)
 y2[1] = y2[0];                                      //update y2(n-1)

 return (y1[0] + y2[0]);                             //add the two tones
}

void main()
{
 comm_intr();                                        //init DSK, codec, McBSP
 while(1);                                           //infinite loop
}
```

图 5.14 产生双音并进行相加的程序（two_tones.c）

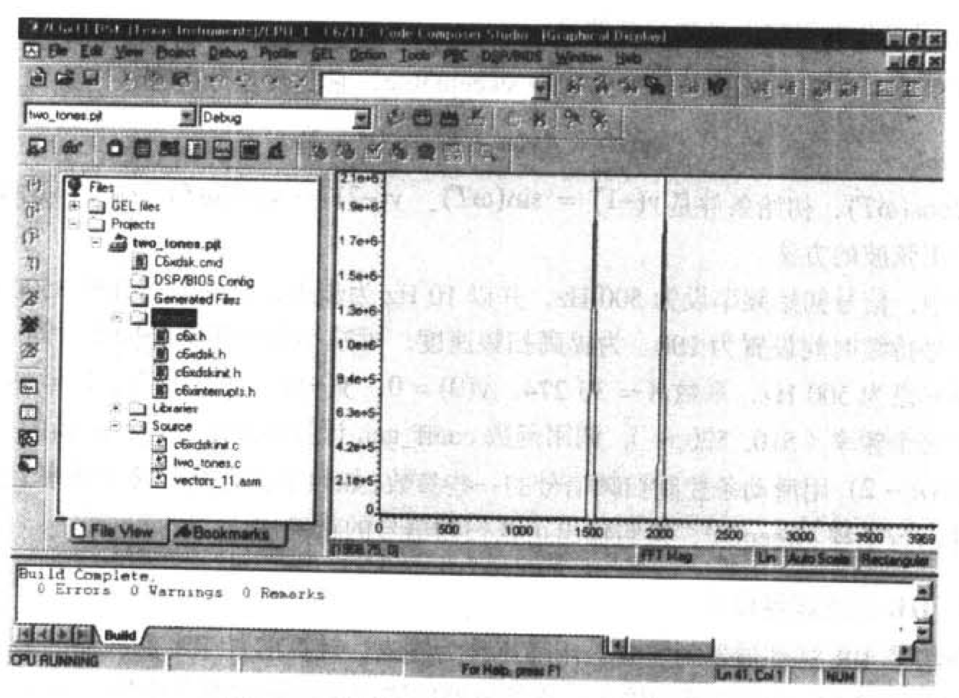


图 5.15 利用 CCS 画出的双音 FFT 幅度谱

图 5.16 给出了 sinegenDE.c 程序，它用另一个差分方程产生正弦波信号。该程序使用实现例 5.2 的另一种方案，在中断服务程序（ISR）内计算差分方程。为了产生 2 kHz 的正弦波，取系数 $A=0$，$y(0)$，$y(1)$和$y(2)$保存在数组 y[3]中。

建立并运行工程 sinegenDE，检验输出是否是一个 2 kHz 的音频信号。将数组和 A 的值改为 y[3] = {0,15 137,11 585}，$A = 12\,540$，重新运行程序，检验输出是否是 1.5 kHz 的音频信号；当 $A=-23\,170$，y[3] = {0,115 85,0}时，输出应为一个 3 kHz 的音频信号。

```
//SinegenDE.c Generates a sinewave using a difference equation

short y[3] = {0,16384,0};                          //y(1) = sinwT
const short A = 0;                                 //A = 2*coswT * 2^14
int n = 2;

interrupt void c_int11()                           //ISR
{
 y[n] = (((int)A*(int)y[n-1])>>14) - y[n-2];       //y(n) = Ay(n-1)-y(n-2)
 y[n-2] = y[n-1];                                  //update y(n-2)
 y[n-1] = y[n];                                    //update y(n-1)
 output_sample(y[n]);                              //output result
 return;                                           //return to main
}

void main()
{
 comm_intr();                                      //init DSK, codec, McBSP
 while(1);                                         //infinite loop
}
```

图 5.16 利用差分方程生成正弦波的程序（sinegenDE.c）

例 5.4　用差分方程产生正弦扫频信号

图 5.17 给出了产生正弦扫频信号的程序 sweepDE.c，它实现差分方程：

$$y(n) = Ay(n-1) - y(n-2)$$

其中$A = 2\cos(\omega T)$，初始条件是$y(-1) = \sin(\omega T)$，$y(-2) = -\sin(2\omega T)$。例 5.2 说明了使用该差分方程产生正弦波的方法。

在程序中，信号初始频率设为 500 Hz，并以 10 Hz 为步进，直到最高频率 3500 Hz，每个频率的正弦信号持续时间设置为 200。为提高扫频速度，可减小每个频率的持续时间。

初始频率设为 500 Hz，系数 $A = 30\,274$，$y(0) = 0$，$y(-1) = -6270$，$y(-2) = -11\,585$（参见例 5.2）。对每个频率（510, 520,…），调用函数 coeff_gen 计算实现差分方程的新的一组系数 A，$y(n-1)$和$y(n-2)$。用滑动条控制扫频信号的一些参数，如频率步进及在每个频率上持续的时间。

建立并运行工程 sweepDE，检验输出正弦扫频信号的正确性。

例 5.5　IIR 逆滤波器设计

该例介绍了 IIR 逆滤波器的实现，用噪声作为输入，计算前置 IIR 滤波器的输出。前置 IIR 滤波器的输出变成 IIR 逆滤波器的输入，逆滤波器的输出是原始的噪声序列。例 4.10 实现了 FIR 逆滤波器，例 5.1 实现了 IIR 滤波器。

IIR 滤波器的传输函数为：

$$H(z) = \frac{\sum_{i=0}^{N-1} a_i z^{-i}}{\sum_{j=1}^{M-1} b_j z^{-j}}$$

IIR 滤波器的输出序列为：

$$y(n) = \sum_{i=0}^{N-1} a_i x(n-i) - \sum_{j=1}^{M-1} b_j y(n-j)$$

这里$x(n-i)$表示输入序列，输入序列$x(n)$可用$\hat{x}(n)$作为$x(n)$的估计，即：

$$\hat{x}(n) = \frac{y(n) + \sum_{j=1}^{M-1} b_j y(n-j) - \sum_{i=1}^{N-1} a_i \hat{x}(n-i)}{a_0}$$

程序 IIRinverse.c（见图 5.18）实现了 IIR 逆滤波器。

建立工程 IIRinverse，用噪声（用噪声发生器或 Goldwave 软件产生）作为输入，运行工程，检验输出结果是否为输入噪声（这时滑动条在默认位置 1）。

把滑动条放在位置 2，检验前置 IIR 滤波器的输出是中心频率为 2 kHz 的带通滤波器，IIR 滤波器是用例 5.1 的系数文件 bp2000.cof 来实现的。当滑动条放在位置 3 时，验证 IIR 逆滤波器的输出是否为原始输入噪声序列。

在该例中，前置滤波器的特性是已知的，可以把它扩展到滤波器特性是未知的情况，这时，前置滤波器的系数 a's 和 b's 可用 Prony 方法估计出来[9]。

```
//SweepDE.c Generates a sweeping sinusoid using a difference equation

#include <math.h>
#define    two_pi    (2*3.1415926)       //2*pi
#define    two_14            16384       //2^14
#define    T              0.000125       //sample period = 1/Fs
#define    MIN_FREQ            500       //initial frequency of sweep
#define    MAX_FREQ           3500       //max frequency of sweep
#define    STEP_FREQ            10       //step frequency
#define    SWEEP_PERIOD        200       //lasting time at one frequency
short      y0 = 0;                       //initial output
short      y_1 = -6270;                  //y(-1)=-sinwT(scaled) f=500Hz
short      y_2 = -11585;                 //y(-2_=-sin2wT(scaled) f=500Hz
short      A = 30274;                    //A = 2*coswT scaled by 2^14
short      freq = MIN_FREQ;              //current frequency
short      sweep_count = 0;              //counter for lasting time
void       coeff_gen(short);             //function prototype for coeff

interrupt void c_int11()                 //ISR
{
 sweep_count++;                          //incr lasting time at one freq
 if(sweep_count >= SWEEP_PERIOD)         //time reaches max duration
  {
   if(freq >= MAX_FREQ)                  //if the current frequency is max
       freq = MIN_FREQ;                  //reinit to initial frequency
   else
       freq = freq + STEP_FREQ;          //incr to next higher frequency

   coeff_gen(freq);                      //function for new set of coeff
   sweep_count = 0;                      //reset counter for lasting time
  }
 y0=(((int)A * (int)y_1)>>14) - y_2;     //y(n) = A*y(n-1) - y(n-2)
 y_2 = y_1;                              //update y(n-2)
 y_1 = y0;                               //update y(n-1)
 output_sample(y0);                      //output result
}

void coeff_gen(short freq)               //calculate new set of coeff
{
 float w;                                //angular frequency

 w = two_pi*freq;                        //w = 2*pi*f
 A = 2*cos(w*T)*two_14;                  //A = 2*coswT * (2^14)
 y_1 = -sin(w*T)*two_14;                 //y_1 = -sinwT *(2^14)
 y_2 = -sin(2*T*w)*two_14;               //y_2 = -sin2wT * (2^14)
 return;
}

void main()
{
 comm_intr();                            //init DSK, codec, McBSP
 while(1);                               //infinite loop
}
```

图 5.17 使用差分方程产生正弦扫频信号的程序（sweepDE.c）

```
//IIRinverse.C  Inverse IIR Filter

#include "bp2000.cof"                 //BP @ 2kHz coefficient file
short dly[stages][2] = {0};           //delay samples per stage
short out_type = 1;                   //type of output for slider
short a0, a1, a2, b1, b2;             //coefficients

interrupt void c_int11()              //ISR
{
 short i, input, input1;
 int un1, yn1, un2, input2, yn2;

 input1 = input_sample();             //input to 1st stage
 input = input1;                      //original input
 for(i = 0; i < stages; i++)          //repeat for each stage
  {
   a1 = ((a[i][1]*dly[i][0])>>15);  //a1*u(n-1)
   a2 = ((a[i][2]*dly[i][1])>>15);  //a2*u(n-2)
   b1 = ((b[i][0]*dly[i][0])>>15);  //b1*u(n-1)
   b2 = ((b[i][1]*dly[i][1])>>15);  //b2*u(n-2)
   un1 = input1 - b1 - b2;
   a0=((a[i][0]*un1)>>15);

   yn1 = a0 + a1 + a2;               //stage output
   input1 = yn1;                     //intermediate out->in next stage
   dly[i][1] = dly[i][0];            //update delays u(n-2) = u(n-1)
   dly[i][0] = un1;                  //update delays u(n-1) = u(n)
  }
 input2 = yn1;                        //out forward=in reverse filter

 for(i = stages; i > 0; i--)          //for inverse IIR filter
  {
   a1 = ((a[i][1]*dly[i][0])>>15);  //a1u(n-1)
   a2 = ((a[i][2]*dly[i][1])>>15);  //a2u(n-2)
   b1 = ((b[i][0]*dly[i][0])>>15);  //b1u(n-1)
   b2 = ((b[i][1]*dly[i][1])>>15);  //b2u(n-2)
   un2 = input2 - a1 - a2;
   yn2 = (un2 + b1 + b2);
   input2 = (yn2<<15)/a[i][0];       //intermediate out->in next stage
  }
 if(out_type == 1)                    //if slider in position 1
    output_sample(input);             //original input signal
 if(out_type == 2)
    output_sample(yn1);               //output of forward filter
 if(out_type == 3)
    output_sample(yn2 >>6);           //output of inverse filter
 return;                              //return from ISR
}

void main()
{
 comm_intr();                         //init DSK, codec, McBSP
 while(1);                            //infinite loop
}
```

图 5.18 实现 IIR 逆滤波的程序（IIRinverse.c）

参考文献

1. L. B. Jackson, *Digital Filters and Signal Processing*, Kluwer Academic, Norwell, MA, 1996.
2. L. B. Jackson, Roundoff noise analysis for fixed-point digital filters realized in cascade or parallel form, *IEEE Transactions on Audio and Electroacoustics*, Vol. AU-18, June 1970, pp. 107–122.
3. L. B. Jackson, An analysis of limit cycles due to multiplicative rounding in recursive digital filters, *Proceedings of the 7th Allerton Conference on Circuit and System Theory*, 1969, pp. 69–78.
4. L. B. Lawrence and K. V. Mirna, A new and interesting class of limit cycles in recursive digital filters, *Proceedings of the IEEE International Symposium on Circuit and Systems*, Apr. 1977, pp. 191–194.
5. R. Chassaing and D. W. Horning, *Digital Signal Processing with the TMS320C25*, Wiley, New York, 1990.
6. A. H. Gray and J. D. Markel, Digital lattice and ladder filter synthesis, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-21, 1973, pp. 491–500.
7. A. H. Gray and J. D. Markel, A normalized digital filter structure, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-23, 1975, pp. 268–277.
8. A. V. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1989.
9. I. Progri, W. R. Michalson, and R. Chassaing, Fast and efficient filter design and implementation on the TMS320C6711 digital signal processor, *International Conference on Acoustics Speech and Signal Processing (ICASSP)*, 2001.
10. N. Ahmed and T. Natarajan, *Discrete-Time Signals and Systems*, Reston Publishing, Reston, VA, 1983.
11. D. W. Horning and R. Chassaing, IIR filter scaling for real-time digital signal processing, *IEEE Transactions on Education*, Feb. 1991.

第 6 章　快速傅里叶变换

本章主要内容包括:（1）基 2 和基 4 的快速傅里叶变换;（2）时域和频域的抽取或分解;（3）编程实例。

快速傅里叶变换（FFT）基于离散傅里叶变换（DFT），它是一种将时域信号转换成等效的频域信号进行运算的高效算法。本章除了介绍其基本原理外，还给出了实时的 FFT 编程实例。

6.1　引言

离散傅里叶变换将时域序列转换成相应的频域序列，而离散傅里叶逆变换进行相反的运算，将频域序列变换成等效的时域序列。快速傅里叶变换基于离散傅里叶变换，但只需要较少的运算，它是一种高效的算法。在数字信号处理中，FFT 是对信号进行频谱分析的最常用算法[1-6]。FFT 运算有两种方法：按时间抽取和按频率抽取。FFT 算法的几个变种：Winograd 变换[7-8]，离散余弦变换（DCT）[9]，离散 Hartley 变换[10-12]也常被使用。基于 DCT，FHT 和 FFT 算法的程序可参见文献[9]。

6.2　基 2 FFT 算法

FFT 算法极大地减少了离散傅里叶变换（DFT）的运算量，离散时间信号$x(nT)$的 DFT 为：

$$X(k)=\sum_{n=0}^{N-1}x(n)W^{nk} \qquad k=0,1,\cdots,N-1 \qquad (6.1)$$

其中抽样周期 T 隐含在$x(n)$中，N 为帧长。常数 W 称为旋转常数或因子，它表示相位，是 N 的函数，即：

$$W=e^{-j2\pi/N} \qquad (6.2)$$

对$k=0,1,\cdots,N-1$，式（6.1）可写为：

$$X(k)=x(0)+x(1)W^{k}+x(2)W^{2k}+\cdots+x(N-1)W^{(N-1)k} \qquad (6.3)$$

这表示一个$N\times N$的矩阵，因为$X(k)$需要计算 N 个值才能得到全部的$X(k)$。由于式（6.3）是复指数的等式，对于每个 k 值，需要$(N-1)$次复数加法和 N 次复数乘法运算，因此共需要(N^2-N)次复数加法和N^2次复数乘法运算。对于 DFT 来说，这就需要非常大的运算量，尤其是在 N 值比较大时，而 FFT 将N^2的运算量变为 $N\log N$，从而大大减少了运算量。

FFT 算法利用了旋转因子的周期性和对称性优点，减少了 FFT 的运算量。由旋转因子的 W 周期性，可得：

$$W^{k+N}=W^{k} \qquad (6.4)$$

由 W 的对称性：

$$W^{k+N/2} = -W^{k} \tag{6.5}$$

图 6.1 给出了$N = 8$时 W 的性质。例如，令$k = 2$，由式（6.4）可得，$W^{10} = W^{2}$，再由式（6.5），可得$W^{6} = -W^{2}$。

对于基 2 的方法，FFT 算法将 N 点 DFT 分解成两个$N/2$点或更短的 DFT，每个$N/2$点的 DFT 再分解成两个$N/4$点的 DFT，该过程一直持续下去，直到最后分解成$N/2$个两点的 DFT，最少的变换点数取决于 FFT 的基。对于基 2 FFT，N 必须是 2 的整数幂，最后分解成两点的 DFT。对于基 4 的 FFT，最后分解成 4 点的 DFT。

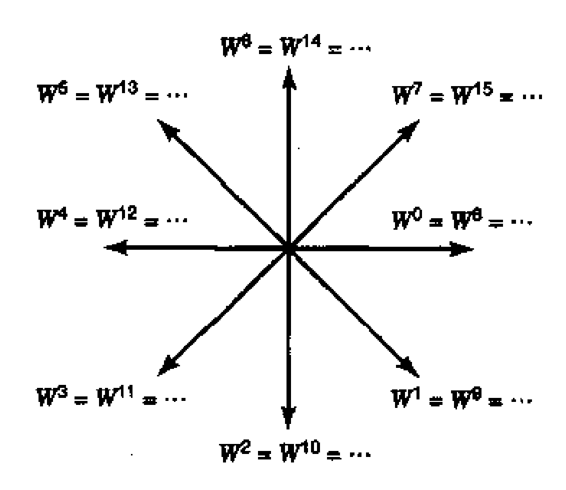


图 6.1　旋转因子 W 的周期性和对称性

6.3 频域抽取的基 2 FFT 算法

将时域输入序列 $x(n)$ 分成两部分：

（a）
$$x(0), x(1), \cdots, x\left(\frac{N}{2}-1\right) \tag{6.6}$$

和

（b）
$$\left(\frac{N}{2}\right), x\left(\frac{N}{2}+1\right), \cdots, x(N-1) \tag{6.7}$$

对式（6.6）和式（6.7）每个序列进行 DFT，可得：

$$X(k) = \sum_{n=0}^{(N/2)-1} x(n)W^{nk} + \sum_{n=N/2}^{N-1} x(n)W^{nk} \tag{6.8}$$

将$n = n + N/2$代入式（6.8）的第二个求和项，有：

$$X(k) = \sum_{n=0}^{(N/2)-1}\left[x(n) - x\left(n+\frac{N}{2}\right)\right]W^{nk} \tag{6.9}$$

其中，由于 $W^{kN/2}$不是 n 的函数，因此可以从第二个求和项中提取出来。利用：

$$W^{kN/2} = e^{-jk\pi} = \left(e^{-j\pi}\right)^{k} = (\cos\pi - j\sin\pi)^{k} = (-1)^{k}$$

代入式（6.9）中，则：

$$X(k)=\sum_{n=0}^{(N/2)-1}\left[x(n)+(-1)^{k}x\left(n+\frac{N}{2}\right)\right]W^{nk} \tag{6.10}$$

因为当k为偶数时，$(-1)^k=1$，而当k为奇数时，$(-1)^k=-1$，将式（6.10）按k的奇偶情况分开，可得：

1．k为偶数时

$$X(k)=\sum_{n=0}^{(N/2)-1}\left[x(n)+x\left(n+\frac{N}{2}\right)\right]W^{nk} \tag{6.11}$$

2．k为奇数时

$$X(k)=\sum_{n=0}^{(N/2)-1}\left[x(n)-x\left(n+\frac{N}{2}\right)\right]W^{nk} \tag{6.12}$$

用$k=2k$代替偶数的k，$k=2k+1$代替奇数的k，式（6.11）和式（6.12）在$k=0,1,\cdots,N/2-1$时可写为：

$$X(2k)=\sum_{n=0}^{(N/2)-1}\left[x(n)+x\left(n+\frac{N}{2}\right)\right]W^{2nk} \tag{6.13}$$

$$x(2k+1)=\sum_{n=0}^{(N/2)-1}\left[x(n)-x\left(n+\frac{N}{2}\right)\right]W^{n}W^{2nk} \tag{6.14}$$

因为旋转因子W是长度N的函数，可表示成W_N，这样W_N^2可写成$W_{N/2}$。令：

$$a(n)=x(n)+x(n+N/2) \tag{6.15}$$

$$b(n)=x(n)-x(n+N/2) \tag{6.16}$$

式（6.13）和式（6.14）可以更简洁地写成两个$N/2$点的DFT，也就是：

$$X(2k)=\sum_{n=0}^{(N/2)-1}a(n)W_{N/2}^{nk} \tag{6.17}$$

$$X(2k+1)=\sum_{n=0}^{(N/2)-1}b(n)W_N^{n}W_{N/2}^{nk} \tag{6.18}$$

图6.2显示了$N=8$时，由N点DFT分解成两个$N/2$点的DFT的过程。从上述的分解过程可以看出，图6.2中$X(k)$的偶数部分在上半部分、奇数部分在下半部分。分解过程可以重复进行下去，这时每$N/2$点的DFT进一步分解成两个$N/4$点的DFT，图6.3再次说明$N=8$的分解过程。

图6.2输出序列上半部分生成图6.3的序列$X(0)$和$X(4)$，表示偶数部分，图6.3的$X(2)$和$X(6)$表示奇数部分。同样，图6.2输出序列的下半部分$X(1)$和$X(5)$，表示偶数部分，而$X(3)$和$X(7)$表示奇数部分，这种顺序打乱是由分解过程造成的。最终输出序列的顺序$X(0),X(4),\cdots$，是打乱的，如图6.3所示，因此输出结果需要重新排序。在后面的章节中，将给出含有适当的排序函数的程序例子。输出序列$X(k)$表示时间序列$x(n)$的DFT。

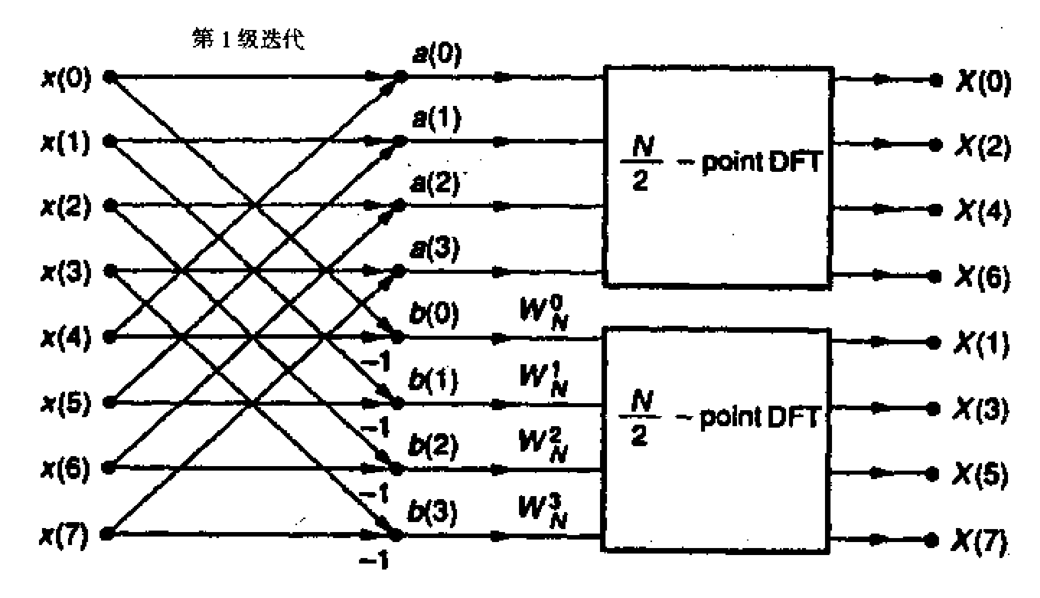


图 6.2　N 点 DFT 分解成两个 $N/2$ 点的 DFT，$N = 8$

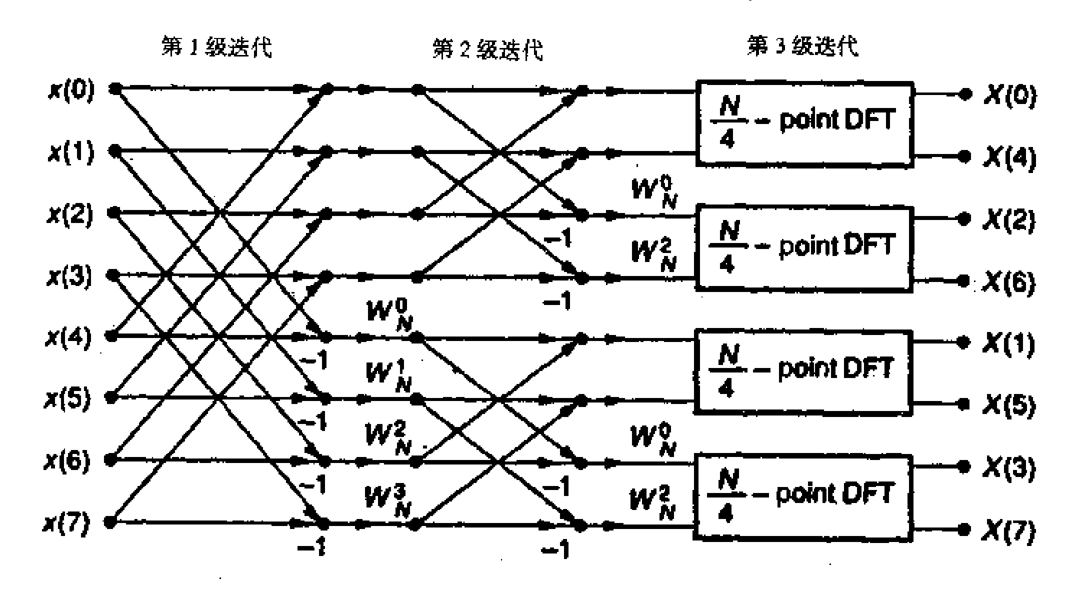


图 6.3　两个 $N/2$ 分解成 4 个 $N/4$ 点的 DFT，$N = 8$

这是最后的分解，因为现在有$N/2$个两点 DFT，对基 2 的 FFT，这是最低分解。对两点的 DFT 来说，式（6.1）中的 $X(k)$ 可表示为：

$$X(k) = \sum_{n=0}^{1} x(n)W^{nk} \qquad k = 0, 1 \tag{6.19}$$

或

$$X(0) = x(0)W^0 + x(1)W^0 = x(0) + x(1) \tag{6.20}$$

$$X(1) = x(0)W^0 + x(1)W^0 = x(0) - x(1) \tag{6.21}$$

因为$W^1 = e^{-j2\pi/2} = -1$，式（6.20）与式（6.21）可表示成图 6.4 所示的流图，通常称为蝶形图。图 6.5 表示了 8 点 FFT 算法的最终流图，这种算法称为频率抽取方法，因为输出序列 $X(k)$ 分解

（抽取）成最小的子序列，这种过程一直继续（迭代）到 M 级，这里 $N = 2^M$。输出 $X(k)$ 是有实部和虚部的复数，FFT 对复数和实数输入序列都适用。

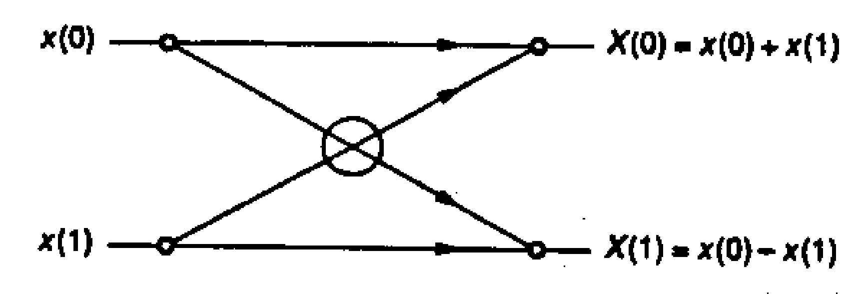


图 6.4 两点的 FFT 蝶形图

FFT 并不是对 DFT 的近似，它和 DFT 的结果是完全一致的，但只需要较少的运算量，而这种运算量的减少对高阶 FFT 运算越来越重要。

还可用其他结构来解释 FFT 算法，图 6.5 的另一种流图是把输入扰乱，输出按自然顺序排列。

下面通过练习说明 8 点的 FFT 算法过程，从图中可以看出，高阶流图（N 较大）也是较容易得到的。

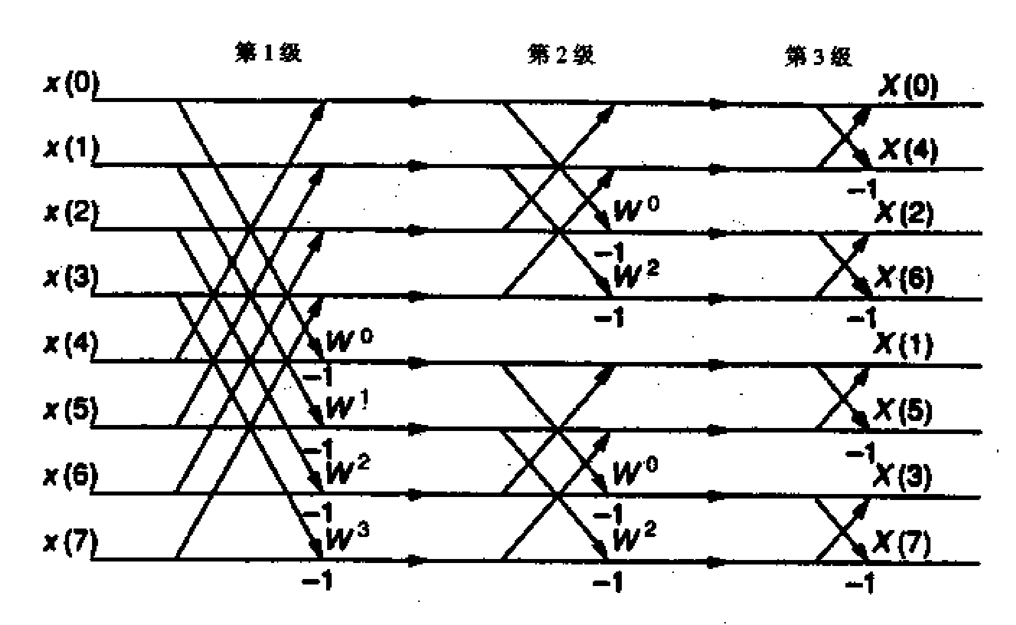


图 6.5 使用频率抽取的 8 点 FFT 流图

练习 6.1 频率抽取法的 8 点 FFT

输入矩形信号 $x(n)$，即 $x(0) = x(1) = x(2) = x(3) = 1$，$x(4) = x(5) = x(6) = x(7) = 0$。利用图 6.5 可以求出输出序列 $X(k)$，$k = 0, 1, \cdots, 7$。对于 $N = 8$，需要计算 4 个旋转因子，即：

$$W^0 = 1$$

$$W^1 = e^{-j2\pi/8} = \cos(\pi/4) - j\sin(\pi/4) = 0.707 - j0.707$$

$$W^2 = e^{-j4\pi/8} = -j$$

$$W^3 = e^{-j6\pi/8} = -0.707 - j0.707$$

在每个运算阶段后，都可求出中间输出序列：

第 1 级迭代

$$
\begin{aligned}
x(0)+x(4) &= 1 \to x'(0) \\
x(1)+x(5) &= 1 \to x'(1) \\
x(2)+x(6) &= 1 \to x'(2) \\
x(3)+x(7) &= 1 \to x'(3) \\
[x(0)-x(4)]W^0 &= 1 \to x'(4) \\
[x(1)-x(5)]W^1 &= 0.707 - j0.707 \to x'(5) \\
[x(2)-x(6)]W^2 &= -j \to x'(6) \\
[x(3)-x(7)]W^3 &= -0.707 - j0.707 \to x'(7)
\end{aligned}
$$

其中$x'(0), x'(1), \cdots, x'(7)$为第 1 级迭代输出的中间序列，也是第 2 级迭代运算的输入序列。

第 2 级迭代

$$
\begin{aligned}
x'(0)+x'(2) &= 2 \to x''(0) \\
x'(1)+x'(3) &= 2 \to x''(1) \\
[x'(0)-x'(2)]W^0 &= 0 \to x''(2) \\
[x'(1)-x'(3)]W^2 &= 0 \to x''(3) \\
x'(4)+x'(6) &= 1-j \to x''(4) \\
x'(5)+x'(7) = (0.707-j0.707)+(-0.707-j0.707) &= -j1.41 \to x''(5) \\
[x'(4)-x'(6)]W^0 &= 1+j \to x''(6) \\
[x'(5)-x'(7)]W^2 &= -j1.41 \to x''(7)
\end{aligned}
$$

第 2 级迭代输出序列为$x''(0), x''(1), \cdots, x''(7)$，也是第 3 级迭代的输入序列。

第 3 级迭代

$$
\begin{aligned}
X(0) &= x''(0)+x''(1) = 4 \\
X(4) &= x''(0)-x''(1) = 0 \\
X(2) &= x''(2)+x''(3) = 0 \\
X(6) &= x''(2)-x''(3) = 0 \\
X(1) &= x''(4)+x''(5) = (1-j)+(-j1.41) = 1-j2.41 \\
X(5) &= x''(4)-x''(5) = 1+j0.41 \\
X(3) &= x''(6)+x''(7) = (1+j)+(-j1.41) = 1-j0.41 \\
X(7) &= x''(6)-x''(7) = 1+j2.41
\end{aligned}
$$

我们现在使用符号 X 作为最后的输出序列，$X(0), X(1), \cdots, X(7)$构成最终扰乱的输出序列，结果可以利用 MATLAB 来验证，参见附录 D。后面我们将介绍怎样进行重新排序以及画出输出幅度谱。

练习 6.2　16 点的 FFT

假定输入矩形序列 $x(0)=x(1)=\cdots=x(7)=1$，$x(8)=x(9)=\cdots=x(15)=0$。利用图 6.6 的 16 点流图，可以求出输出序列。每级的中间输出结果可用练习 6.1 类似的方法求出。$N=16$时，需要计算 8 个旋转因子$W^0, W^1, \cdots, W^7$。

检验如图 6.6 所示的扰乱输出序列 X，重新对它们进行排序，求出幅度谱。检验图 6.7 所示的图形，它是一个 sinc 函数。输出$X(8)$ 表示在奈奎斯特频率处的幅度值。这些结果可用 MATLAB 进行校验，参见附录 D。

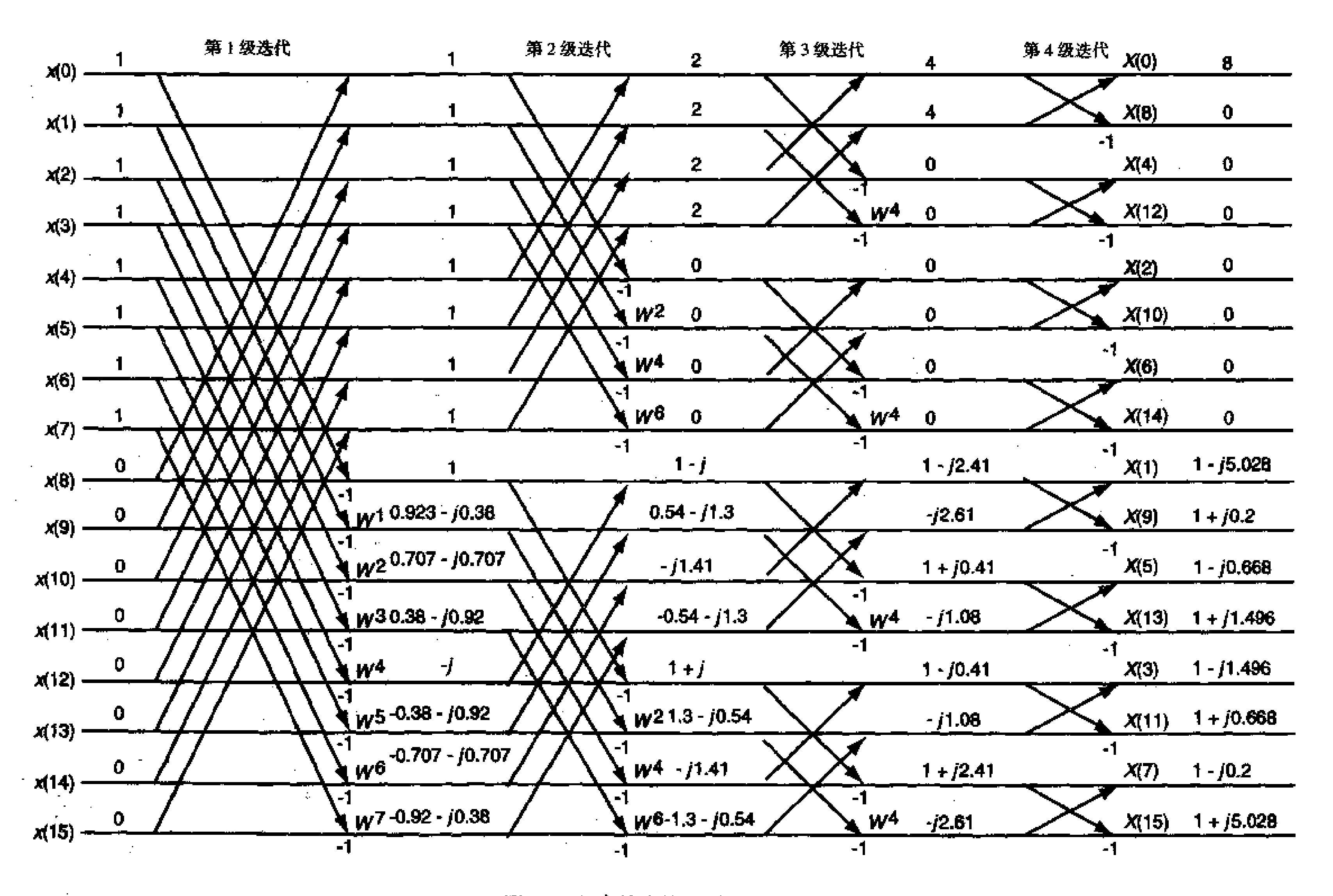


图6.6 频率抽取的16点FFT流图

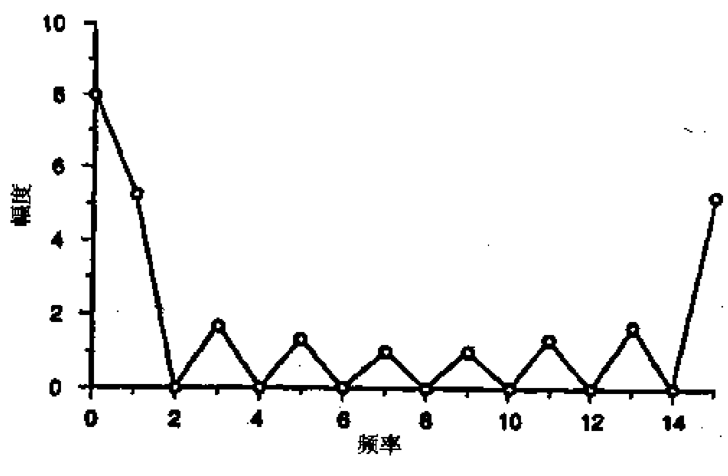


图 6.7 16点FFT幅度谱

6.4 时间抽取的基2 FFT 算法

频率抽取法（DIF）是将输出序列分解成较小的子序列，而时间抽取法（DIT）却是将输入序列分解成较小的子序列。DIT 将输入序列分解成偶数和奇数两个序列，即：

$$x(0), x(2), x(4), \cdots, x(2n)$$

和

$$x(1), x(3), x(5), \cdots, x(2n+1)$$

代入式（6.1），可得：

$$X(k)=\sum_{n=0}^{(N/2)-1} x(2n)W^{2nk}+\sum_{n=0}^{(N/2)-1} x(2n+1)W^{(2n+1)k} \tag{6.22}$$

将 $W_N^2=W_{N/2}$ 代入式（6.22）中，可得：

$$X(k)=\sum_{n=0}^{(N/2)-1} x(2n)W_{N/2}^{nk}+W_N^k\sum_{n=0}^{(N/2)-1} x(2n+1)W_{N/2}^{nk} \tag{6.23}$$

它代表两 个$N/2$ 点的 DFT。令：

$$C(k)=\sum_{n=0}^{(N/2)-1} x(2n)W_{N/2}^{nk} \tag{6.24}$$

$$D(k)=\sum_{n=0}^{(N/2)-1} X(2n+1)W_{N/2}^{nk} \tag{6.25}$$

则式（6.23）中$X(k)$可以表示为：

$$X(k)=C(k)+W_N^kD(k) \tag{6.26}$$

为了满足$k>(N/2)-1$的要求，式（6.26）需要进行变换。利用式（6.5）旋转常数的对称性，即$W^{k+N/2}=-W^k$，可得：

$$X(k+N/2)=C(k)-W^kD(k) \qquad k=0,1,\cdots,(N/2)-1 \tag{6.27}$$

例如，对$N=8$，式（6.26）和式（6.27）变成：

$$X(k)=C(k)+W^kD(k) \qquad k=0,1,2,3 \tag{6.28}$$

$$X(k+4) = C(k) - W^k D(k) \qquad k = 0, 1, 2, 3 \tag{6.29}$$

图 6.8 显示了用时间抽取方法将 8 点的 DFT 分解成两个 4 点的 DFT 过程，分解或抽取过程重复进行下去，直到每 4 个点的 DFT 分解成两个 2 点的 DFT，如图 6.9 所示，随着分解的继续，最后分解为$N/2$个 2 点的 DFT。

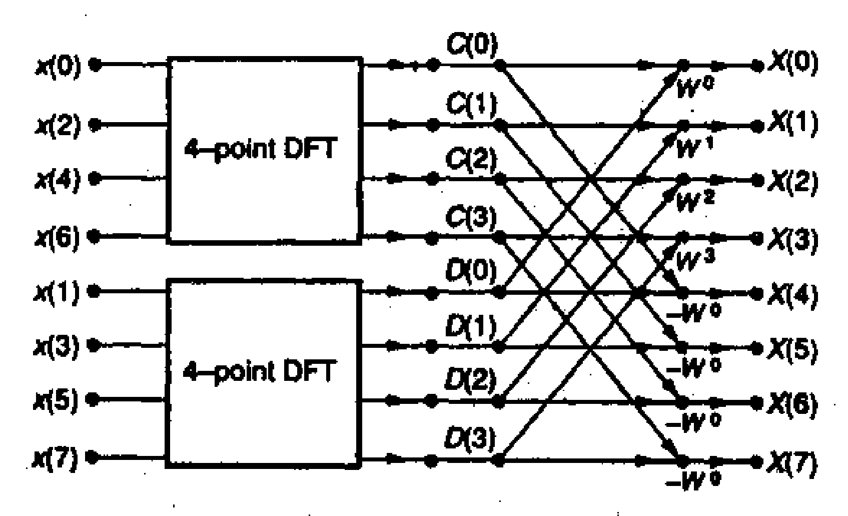


图 6.8　使用 DIT，8 个点的 DFT 分解成 4 点 DFT

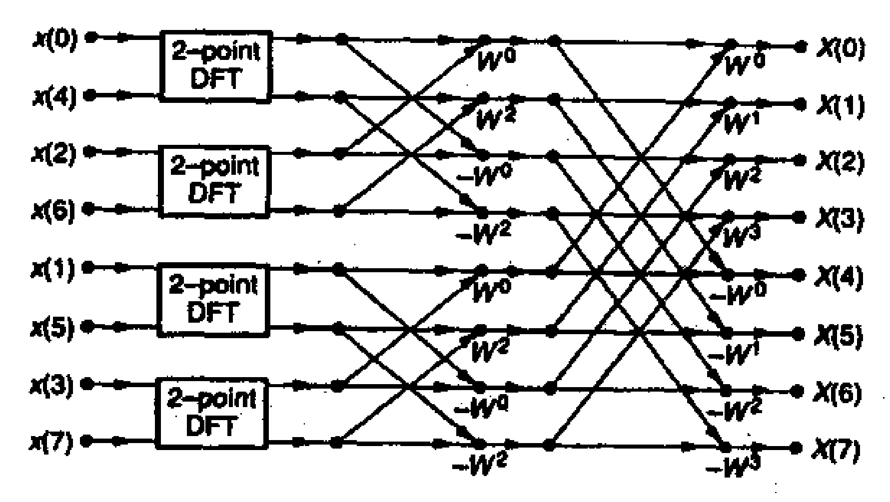


图 6.9　使用 DIT，两个 4 点的 DFT 分解成 4 个 2 点 DFT

图 6.10 显示了按时间抽取的 8 点 FFT 的流图，从图 6.10 中可以看到输入序列是扰乱的，这与按频率抽取得到的输出序列$X(k)$的扰乱方式是一样的。当输入序列$x(n)$扰乱后，输出序列$X(k)$就变成了自然顺序。采用按时间抽取 DIT 或按频率抽取 DIF 的方法来计算 FFT 得到的结果都是一样的，图 6.10 给出了另一种 DIT 流图，输入序列是自然序列，而输出序列是扰乱的。

下面的练习说明了采用 DIT 方法计算 8 点 FFT 的过程，计算结果与练习 6.1 用 DIF 方法计算的结果是一致的。

练习 6.3　按时间抽取的 8 点 FFT

假定使用和练习 6.1 中一样的矩形序列$x(n)$作为输入序列，利用图 6.10 的 DIT 流图，得到与练习 6.1 相同的输出序列$X(k)$。旋转常数与练习 6.1 是一样的，但注意旋转常数 W 只和第二项相乘（而不是第一项）。

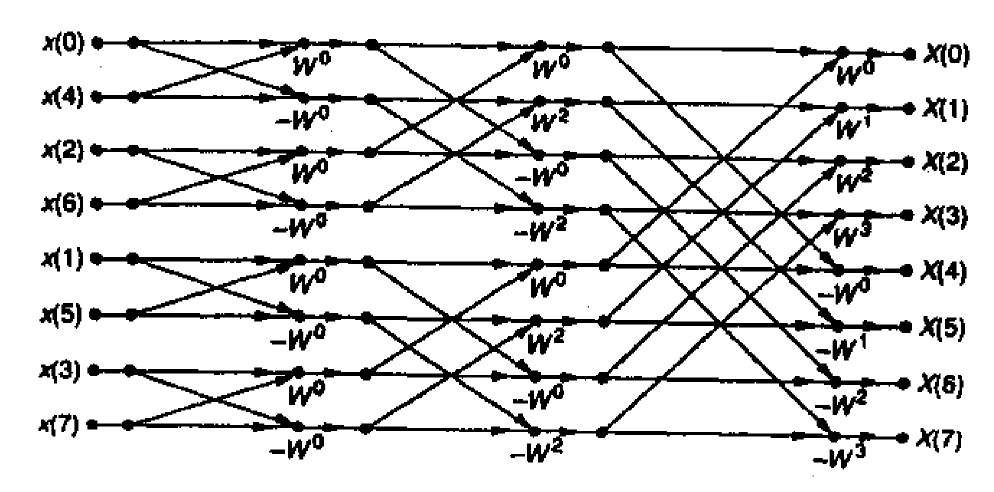


图 6.10 时间抽取 8 点的 FFT 流图

第 1 级迭代

$$x(0) + W^0x(4) = 1 + 0 = 1 \rightarrow x'(0)$$
$$x(0) - W^0x(4) = 1 - 0 = 1 \rightarrow x'(4)$$
$$x(2) + W^0x(6) = 1 + 0 = 1 \rightarrow x'(2)$$
$$x(2) - W^0x(6) = 1 - 0 = 1 \rightarrow x'(6)$$
$$x(1) + W^0x(5) = 1 + 0 = 1 \rightarrow x'(1)$$
$$x(1) - W^0x(5) = 1 - 0 = 1 \rightarrow x'(5)$$
$$x(3) + W^0x(7) = 1 + 0 = 1 \rightarrow x'(3)$$
$$x(3) - W^0x(7) = 1 - 0 = 1 \rightarrow x'(7)$$

其中序列 x 表示第 1 级迭代的输出序列，也是第 2 级迭代的输入序列。

第 2 级迭代

$$x'(0) + W^0x'(2) = 1 + 1 = 2 \rightarrow x''(0)$$
$$x'(4) + W^2x'(6) = 1 + (-j) = 1 - j \rightarrow x''(4)$$
$$x'(0) - W^0x'(2) = 1 - 1 = 0 \rightarrow x''(2)$$
$$x'(4) - W^2x'(6) = 1 - (-j) = 1 + j \rightarrow x''(6)$$
$$x'(1) + W^0x'(3) = 1 + 1 = 2 \rightarrow x''(1)$$
$$x'(5) + W^2x'(7) = 1 + (-j)(1) = 1 - j \rightarrow x''(5)$$
$$x'(1) - W^0x'(3) = 1 - 1 = 0 \rightarrow x''(3)$$
$$x'(5) - W^2x'(7) = 1 - (-j)(1) = 1 + j \rightarrow x''(7)$$

其中序列 x'' 是第 2 级迭代的输出序列，也是第 3 级迭代的输入序列。

第 3 级迭代

$$X(0) = x''(0) + W^0x''(1) = 4$$
$$X(1) = x''(4) + W^1x''(5) = 1 - j2.414$$
$$X(2) = x''(2) + W^2x''(3) = 0$$
$$X(3) = x''(6) + W^3x''(7) = 1 - j0.414$$
$$X(4) = x''(0) - W^0x''(1) = 0$$
$$X(5) = x''(4) - W^1x''(5) = 1 + j0.414$$
$$X(6) = x''(2) - W^2x''(3) = 0$$
$$X(7) = x''(6) - W^3x''(7) = 1 + j2.414$$

可见，其输出序列与练习 6.1 是相同的。

6.5 位反转排序方法

位反转过程是对扰乱的序列进行重新排序。为了证明这种方法的正确性，取 $N=8$，用 3 个位表示，将首末位进行反转。例如，$(100)_b$ 反转为 $(001)_b$，因此 $(100)_b$ 作为 $X(4)$ 的地址，反转成 $X(1)$ 的地址 $(001)_b$。同样地，$(110)_b$ 反转为$(001)_b$，即 $X(6)$ 的地址与 $X(3)$ 的地址进行互换，图 6.5 DIF 的输出序列和图 6.10 DIT 的输入序列可按这种方法重新进行排序。

位反转方法可应用于 N 值比较大时，例如$N=64$可用 6 个位表示，只需将第 1 个和第 6 个位相互反转，第 2 个和第 5 个位相互反转以及第 3 个和第 4 个相互反转即可。

本章将举几个使用这种排序算法的例子，说明 FFT 算法规则。

6.6 基 4 FFT 算法

基 4 算法能提高 FFT 的运算速度。关于 FFT 算法，已经开发了按高阶基和分裂基 FFT 算法程序。我们用频率抽取法 DIF 的分解过程来介绍基 4 FFT 算法。基 4 算法的最后或最低的分解包含 4 个输入和 4 个输出。FFT 的阶数或长度是4^M，M 是级数。例如，对 16 点的 FFT，只需要两级，而对基 2 算法来说就需要 4 级。式（6.1）的 DFT 分解成 4 项的和，而不是两项的和，如式（6.30）所示：

$$X(k)=\sum_{n=0}^{(N/4)-1}x(n)W^{nk}+\sum_{n=N/4}^{(N/2)-1}x(n)W^{nk}+\sum_{n=N/2}^{(3N/4)-1}x(n)W^{nk}+\sum_{n=3N/4}^{N-1}x(n)W^{nk} \qquad (6.30)$$

分别用$n=n+N/4$、$n=n+N/2$、$n=n+3N/4$来代替后三个和式中的 n，则式（6.30）可以表示为：

$$\begin{aligned}X(k)=&\sum_{n=0}^{(N/4)-1}x(n)W^{nk}+W^{kN/4}\sum_{n=0}^{(N/4)-1}x(n+N/4)W^{nk}\\&+W^{kN/2}\sum_{n=0}^{(N/4)-1}x(n+N/2)W^{nk}+W^{3kN/4}\sum_{n=0}^{(N/4)-1}x(n+3N/4)W^{nk}\end{aligned} \qquad (6.31)$$

它表示 4 个$N/4$点的 DFT。利用：

$$W^{kN/4}=\left(e^{-j2\pi/N}\right)^{kN/4}=e^{-jk\pi/2}=(-j)^k$$
$$W^{kN/2}=e^{-jk\pi}=(-1)^k$$
$$W^{3kN/4}=(j)^k$$

式（6.31）变成：

$$X(k)=\sum_{n=0}^{(N/4)-1}\left[x(n)+(-j)^k x(n+N/4)+(-1)^k x(n+N/2)+(j)^k x(n+3N/4)\right]W^{nk} \qquad (6.32)$$

令$W_N^4=W_{N/4}$，式（6.32）可写为：

$$X(4k)=\sum_{n=0}^{(N/4)-1}[x(n)+x(n+N/4)+x(n+N/2)+x(n+3N/4)]W_{N/4}^{nk} \qquad (6.33)$$

$$X(4k+1)=\sum_{n=0}^{(N/4)-1}[x(n)-jx(n+N/4)-x(n+N/2)+jx(n+3N/4)]W_N^nW_{N/4}^{nk} \qquad (6.34)$$

$$X(4k+2)=\sum_{n=0}^{(N/4)-1}[x(n)-x(n+N/4)+x(n+N/2)-x(n+3N/4)]W_N^{2n}W_{N/4}^{nk} \tag{6.35}$$

$$X(4k+3)=\sum_{n=0}^{(N/4)-1}[x(n)+jx(n+N/4)-x(n+N/2)-jx(n+3N/4)]W_N^{3n}W_{N/4}^{nk} \tag{6.36}$$

其中$k=0,1,\cdots,(N/4)-1$。式（6.33）到式（6.36）表示生成 4 个 4 点 DFT 的分解过程。图 6.11 给出了基 4 的 16 点频域抽取的 FFT 流图。注意流图中的 4 点蝶形图。$\pm j$ 和-1 在图 6.11 中没有显示出来，流图的结果用于下面的练习中。

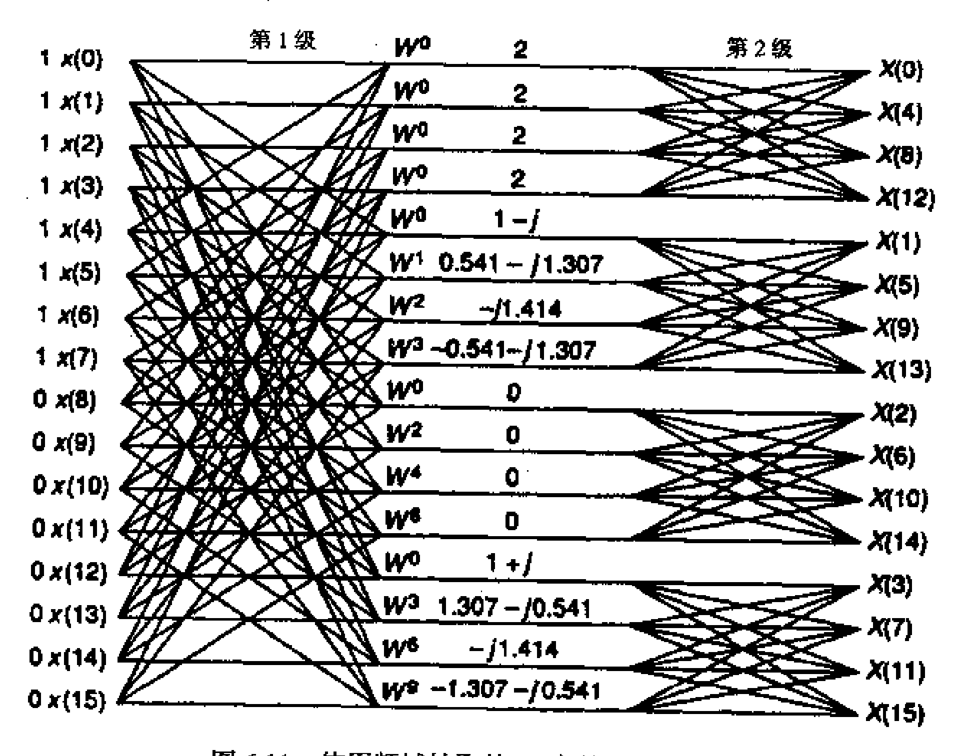


图 6.11　使用频域抽取的 16 点基 4 FFT 流图

练习 6.4　基 4 的 16 点 FFT

给定如练习 6.2 的一个矩形输入序列$x(n)$，序列$x(0)=x(1)=\cdots=x(7)=1$，$x(8)=x(9)=\cdots=x(15)=0$，利用图 6.11 的基 4 算法流图求出 16 点 FFT 的输出序列，所用到的旋转常数如表 6.1 所示。

表 6.1　基 4 的 16 点 FFT 旋转常数

m	W_N^m	$W_{N/4}^m$
0	1	1
1	0.9238 − j0.3826	−j
2	0.707 − j0.707	−1
3	0.3826 − j0.9238	+j
4	0 − j	1
5	−0.3826 − j0.9238	−j
6	−0.707 − j0.707	−1
7	−0.9238 − j0.3826	+j

第 1 级迭代的输出序列如图 6.11 所示。例如下面的输出序列：

$$[x(0)+x(4)+x(8)+x(12)]W^0=1+1+0+0=2\to x'(0)$$
$$[x(1)+x(5)+x(9)+x(13)]W^0=1+1+0+0=2\to x'(1)$$
$$\vdots$$
$$[x(0)-jx(4)-x(8)+jx(12)]W^0=1-j-0-0=1-j\to x'(4)$$
$$\vdots$$
$$[x(3)-x(7)+x(11)-x(15)]W^6=0\to x'(11)$$
$$[x(0)+jx(4)-x(8)-jx(12)]W^0=1+j-0-0=1+j\to x'(12)$$
$$\vdots$$
$$[x(3)+jx(7)-x(11)-jx(15)]W^9=[1+j-0-0](-W^1)$$
$$=-1.307-j0.541\to x'(15)$$

第 2 级迭代之后的输出序列如下所示：

$$X(3)=(1+j)+(1.307-j0.541)+(-j1.414)+(-1.307-j0.541)=1-j1.496$$

和

$$X(15)=(1+j)(1)+(1.307-j0.541)(-j)+(-j1.414)(1)$$
$$+(-1.307-j0.541)(-j)=1+j5.028$$

输出序列$X(0), X(1),\cdots, X(15)$和图 6.6 中基 2 的 16 点 FFT 输出结果是一样的，同样可以利用 MATLAB 来检验输出结果，参见附录 D。

输出序列次序扰乱了，需要进行重新排序。和基 2 运算中的位反转一样，排序可用数字反转方法来实现。基 4（4 为基）用数字 0, 1, 2, 3。如$X(8)$和$X(2)$的地址需要交换，因为以 10 为基或者说十进制的 $(8)_{10}$ 相当于以 4 为基的 $(20)_4$，数字 0 和 1 位置调换就变成以 4 为基的$(02)_4$，相当于十进制的 $(02)_{10}$。

虽然混合或者更高的基在计算上可以更简单，但编程问题也变得比较复杂，所以基 2 算法仍被广泛地使用，其次是基 4 算法。

6.7 快速傅里叶逆变换

离散傅里叶逆变换（IDFT）将频域中序列$X(k)$转换成相应的时间序列$x(n)$。定义为：

$$x(n)=\frac{1}{N}\sum_{k=0}^{N-1}X(k)W^{-nk}\qquad n=0,1,\cdots,N-1 \tag{6.37}$$

将式（6.37）和式（6.1）的 DFT 公式进行比较，可以看出前面介绍的 FFT 算法，只要改变两个地方，就可用来求快速傅里叶逆变换（IFFT）：

1. 增加比例因子$1/N$；
2. 用W^{nk}的复共轭W^{-nk}代替W^{nk}。

通过这些修改，可用同样的 FFT 流图来求快速傅里叶逆变换。下面同样用程序例子来说明 FFT 的逆变换。

FFT 算法的变种，如快速哈特雷算法（FHT），可以很容易从 FFT 算法推导出来。反过来，FFT 算法也能从 FHT 算法推导出来。有关快速哈特雷变换算法流图的画法以及 8 点和 16 点的快速哈特雷变换，可参见文献[12]。

练习 6.5　8 点 IFFT

将练习 6.1 中得到的输出序列$X(0) = 4, X(1) = 1 - j2.41, \cdots, X(7) = 1 + j2.41$ 作为 8 点 IFFT 流图的输入序列，再做上述的两个修改（增加定标系数和 W 的复共轭），由 8 点的 FFT 流图（前向）得到与图 6.5 相似的 8 点 IFFT（反向）流图。检验输出序列是否为练习 6.1 的输入矩形序列$x(0) = 1, x(1) = 1, \cdots, x(7) = 0$。

6.8　编程举例

例 6.1　用 CCS 窗口显示实数序列的 DFT

该例说明了 N 点序列的离散傅里叶变换 DFT，图 6.12 给出了实现 DFT 的程序 DFT.c。输入序列为$x(n)$，程序按照下式进行计算：

$$X(k) = \mathrm{DFT}\{x(n)\} = \sum_{n=0}^{N-1} x(n)W^{nk} \qquad k = 0, 1, \cdots, N-1$$

其中旋转因子$W = e^{-j2\pi/N}$。上式可分解成实部和虚部分别计算，即：

$$\mathrm{Re}\{X(k)\} = \sum_{n=0}^{N-1} x(n)\cos(2\pi nk/N)$$

$$\mathrm{IM}\{X(k)\} = \sum_{n=0}^{N-1} x(n)\sin(2\pi nk/N)$$

使用具有整数周期 m 的实数序列，对于所有的 k，除了 $k = m$和 $k = N - m$点外，$X(k)$ 不为 0，其他$X(k) = 0$。

建立工程 DFT，输入序列$x(n)$为$N = 8$个数据点的余弦。为了检验输出结果，进行如下操作：

1. 选择菜单 View→Watch Window，插入 j 和 out 两个表达式（右击 Watch 窗口）。点击+out，打开窗口，观察分别表示实部和虚部的 out[0]和 out[1]。
2. 在 DFT 函数调用后面的括号“}”处设置一个断点。
3. 选择菜单 Debug→Animate（其速度可利用选项调整）。检验实部值 out[0] 在 $j = 1$ 时，它的值大（为 3996），而在$j = 7$ 时，它的值小。因为 $m = 1$，且$x(n)$是单周期序列。点数 $N = 8$，尖峰出现在$j = m = 1$ 和 $j = N - m = 7$ 处。下面两个 MATLAB 命令可以检验这些结果（参见附录 D）：

   ```
   x = [1000 707 0 -707 -1000 -707 0 707];
   y = fft(x)
   ```

   注意表中的数据是有舍入和舍出的（尖峰最大值是 3996，不是 4000）。因为是余弦信号，虚部 out[1]为 0（很小）。在实际应用中，通常 F_s=8 kHz，生成的信号频率是$f = F_s \times$周期数/N = 1 kHz。
4. 使用有 20 个数据点两个周期的正弦数据表作为输入$x(n)$。在程序内，把 N 改成等于 20，用注释语句屏蔽掉余弦表（第一次输入），改用正弦表作为输入。重建程序，再进行仿真。检验当$j = 2$ 时，有一个较大的负值（−10 232）；而在$j = N - m$ =18 时，有一个较大的正值（10 232）。实际应用中，可求出 $k = 0, 1, \cdots$时 $X(k)$的模。当F_s = 8 kHz时，生成对应的信号频率f = 800 Hz。

```
//DFT.c DFT of N-point from lookup table. Output from watch window

#include <stdio.h>
#include <math.h>
void dft(short *x, short k, int *out); //function prototype
#define N 8                           //number of data values
float pi = 3.1416;

short x[N] = {1000,707,0,-707,-1000,-707,0,707}; //1-cycle cosine

//short x[N]={0,602,974,974,602,0,-602,-974,-974,-602,
//           0,602,974,974,602,0,-602,-974,-974,-602}; //2-cycles sine

int out[2] = {0,0};                   //init Re and Im results

void dft(short *x, short k, int *out) //DFT function
{
 int sumRe = 0;                       //init real component
 int sumIm = 0;                       //init imaginary component
 int i = 0;
 float cs = 0;                        //init cosine component
 float sn = 0;                        //init sine component

 for (i = 0; i < N; i++)              //for N-point DFT
   {
     cs = cos(2*pi*(k)*i/N);          //real component
     sn = sin(2*pi*(k)*i/N);          //imaginary component
     sumRe = sumRe + x[i]*cs;         //sum of real components
     sumIm = sumIm - x[i]*sn;         //sum of imaginary components
   }
 out[0] = sumRe;                      //sum of real components
 out[1] = sumIm;                      //sum of imaginary components
}
void main()
{
 int j;

 for (j = 0; j < N; j++)
   {
     dft(x,j,out);                    //call DFT function
   }
}
```

图 6.12 用查找表数据作为输入的 DFT 实现程序（DFT.c）

例 6.2 利用 C 语言 FFT 函数实现实时输入信号的 FFT

图 6.13 给出了实现外部输入信号的 256 点 FFT 程序 FFT256c.c，它调用了通用 FFT 的 C 语言函数 FFT.c（在辅助材料中）。文献 13 和文献 14 介绍了这个和 C31 DSK 以及 C30 EVM 一起使用的 FFT 函数。

```
//FFT256c.c FFT implementation calling C-coded FFT function

#include <math.h>
#define PTS 256                       //# of points for FFT
#define PI 3.14159265358979
typedef struct {float real,imag;} COMPLEX;
void FFT(COMPLEX *Y, int n);          //FFT prototype
float iobuffer[PTS];                  //as input and output buffer
float x1[PTS];                        //intermediate buffer
short i;                              //general purpose index variable
short buffercount = 0;                //number of new samples in iobuffer
short flag = 0;                       //set to 1 by ISR when iobuffer full
COMPLEX w[PTS];                       //twiddle constants stored in w
COMPLEX samples[PTS];                 //primary working buffer

main()
{
 for (i = 0 ; i<PTS ; i++)            // set up twiddle constants in w
  {
   w[i].real = cos(2*PI*i/512.0);     //Re component of twiddle constants
   w[i].imag =-sin(2*PI*i/512.0);     //Im component of twiddle constants
  }
 comm_intr();                         //init DSK, codec, McBSP

 while(1)                             //infinite loop
  {
   while (flag == 0) ;                //wait until iobuffer is full
   flag = 0;                          //reset flag
   for (i = 0 ; i < PTS ; i++)        //swap buffers
    {
     samples[i].real=iobuffer[i];     //buffer with new data
     iobuffer[i] = x1[i];             //processed frame to iobuffer
    }
   for (i = 0 ; i < PTS ; i++)
     samples[i].imag = 0.0;           //imag components = 0

   FFT(samples,PTS);                  //call function FFT.c

   for (i = 0 ; i < PTS ; i++)        //compute magnitude
    {
     x1[i] = sqrt(samples[i].real*samples[i].real
           + samples[i].imag*samples[i].imag)/32;
    }
   x1[0] = 32000.0;                   //negative spike(with AD535)for ref
  }                                   //end of infinite loop
}                                     //end of main

interrupt void c_int11()              //ISR
{
  output_sample((int)(iobuffer[buffercount]));      //out from iobuffer
  iobuffer[buffercount++]=(float)(input_sample());  //input to iobuffer
  if (buffercount >= PTS)                           //if iobuffer full
  {
      buffercount = 0;                              //reinit buffercount
      flag = 1;                                     //set flag
  }
}
```

图 6.13 调用 C 语言 FFT 函数的实时输入 FFT 程序（FFT256c.c）

旋转常数 W 在程序内计算。为了说明实现过程，输入信号的虚部设成 0。由 FFT 结果得到幅度谱（经过定标），这些值再输出到编解码器。在此过程中，使用了下面三个存储缓冲区：

1. samples，存储需要变换的数据；
2. iobuffer，接收输入抽样数据并输出处理后的数据；
3. x1，存储经 FFT 处理并经过定标的幅度谱值。

每个抽样周期产生一次中断。在每次中断时，缓冲区 iobuffer 输出一个数据给编解码器的 DAC，同时将一个新输入数据存到同样的缓冲区中。该缓冲区的指针（buffercount）当做标志，表示它是否已满。如果该缓冲区是满的，数据将被复制到另一个缓冲区 samples 中，这些数据在调用 FFT 函数时需要用到。保存在缓冲区 x1、已经 FFT 处理及定标后的幅度谱值现在可以复制到 I/O 缓冲区 iobuffer 中作为输出。在滤波算法中，每采集一个新数据就要进行一次处理，但在 FFT 算法中，只有当 FFT 算法要求的整帧数据全部到来时，才能进行 FFT 处理。

建立并运行工程 FFT256c，输入一个幅度近似为 0.5 V 到 1 V 之间的 2 kHz 正弦信号，图 6.14 给出了经 FFT 变换后幅度谱的时域波形，该波形是利用 HP 动态信号分析仪得到的（也可以使用示波器）。从图 6.14 可以看出：两个负尖峰相隔 $256T_s = 32$ ms，它也表示抽样频率F_s。第一个正尖峰对应 2 kHz 处（在对应 4 kHz 的两个尖峰中间），第二个正尖峰对应于折叠频率$F_s - f = 6$ kHz。增大输入信号频率，观察两个尖峰向 4 kHz 的奈奎斯特频率处靠拢。

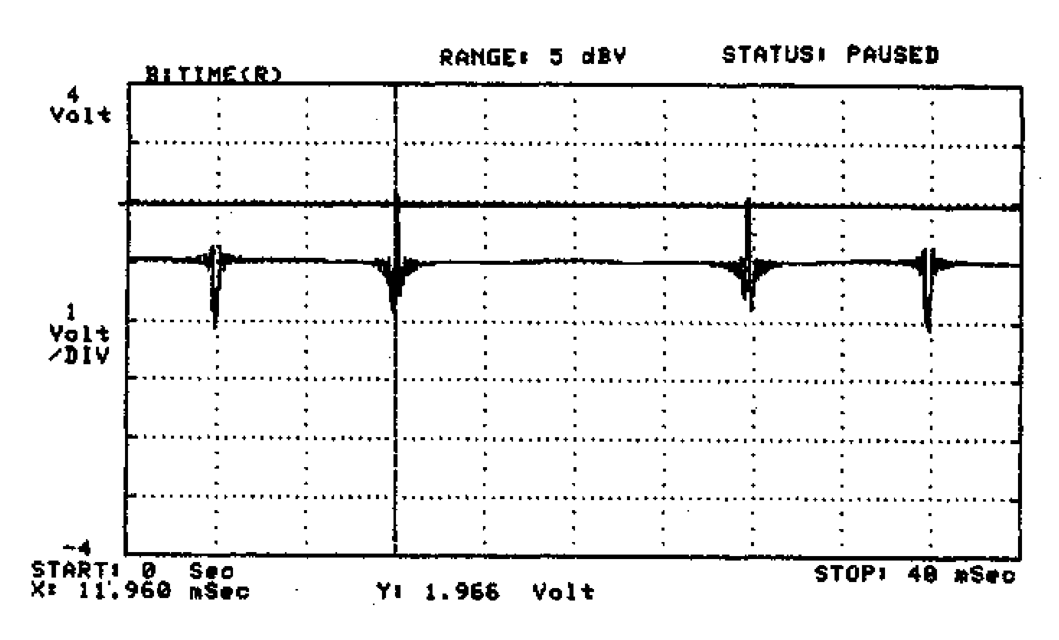


图 6.14 表示实时输入 FFT 幅度谱的时域图

练习 6.3 使用 TI 的 C 可调用 FFT 函数，求由表输入的正弦信号的 FFT

图 6.15 给出了 FFTsinetable.c 程序，该程序说明了如何用 C 程序调用 TI 公司的浮点 FFT 函数 cfftr2_dit.sa，该函数可从 TI 公司的网站上获得。旋转常数是在程序内计算的。按照 FFT 函数的要求，旋转常数的虚部要进行取反。FFT 函数同时假定有 $N/2$ 个复旋转常数。在存储缓冲区中，数据对齐（以 8 个字节为单位）是非常重要的，输入数据和旋转常数都是复数。

表格中的正弦数据作为输入信号实部，虚部设为 0。根据 FFT 函数的要求，输入数据在存储器中连续排成实部和虚部数对，输出结果也是复数。

```
//FFTsinetable.c FFT(sine)from table. Calls TI float-point FFT function

#include <math.h>
#define N 32                                //number of FFT points
#define SQRT_N 32                           //SQRT_N >= SQRT(N)
#define FREQ 8                              //# of points/cycle
#define RADIX 2                             //radix or base
#define DELTA (2*PI)/N                      //argument for sine/cosine
#define TAB_PTS 32                          //# of points in sine_table
#define PI 3.14159265358979
short i = 0;
short iTwid[SQRT_N];                        //N/2 + 1 > sqrt(N)
short iData[N];                             //index for bitrev X
float Xmag[N];                              //magnitude spectrum of x
typedef struct Complex_tag {float re,im;}Complex;
Complex W[N/RADIX];                         //array for twiddle constants
Complex x[N];                               //N complex data values
#pragma DATA_ALIGN(W,sizeof(Complex))       //align boundary size complex

short sine_table[TAB_PTS] = {0,195,383,556,707,831,924,981,1000,
981,924,831,707,556,383,195,-0,-195,-383,-556,-707,-831,-924,-981,
-1000,-981,-924,-831,-707,-556,-383,-195};

void main()
{
 for( i = 0 ; i < N/RADIX ; i++ )
  {
   W[i].re = cos(DELTA*i);                  //real component of W
   W[i].im = sin(DELTA*i);                  //neg imag component
  }                                         //see cfftr2_dit
 for( i = 0 ; i < N ; i++ )
  {
   x[i].re=3*sine_table[FREQ*i % TAB_PTS]; //wrap when i=TAB_PTS
   x[i].im = 0 ;                            //zero imaginary part
  }
 digitrev_index(iTwid, N/RADIX, RADIX);     //produces index for bitrev() W
 bitrev(W, iTwid, N/RADIX);                 //bit reverse W

 cfftr2_dit(x, W, N ) ;                     //TI floating-pt complex FFT

 digitrev_index(iData, N, RADIX);           //produces index for bitrev() X
 bitrev(x, iData, N);                       //freq scrambled->bit-reverse X
 for(i = 0 ; i < N ; i++ )
   Xmag[i] = sqrt(x[i].re*x[i].re+x[i].im*x[i].im ); //magnitude of X

 comm_poll( ) ;                             //init DSK,codec,McBSP
 while (1)                                  //infinite loop
  {
   output_sample(32000) ;                   //negative spike as reference
   for (i = 1; i < N; i++)
     output_sample((int)Xmag[i]);           //output magnitude samples
  }
}
```

图6.15 使用TI优化的浮点复FFT函数和表作为输入数据的FFT程序（FFTsinetable.c）

FFT 函数 cfftr2_dit.sa 是复输入信号时间抽取基 2 的 FFT 算法程序。两个辅助函数 digitrev_index.c、bitrev.sa 和复 FFT 函数一起使用，用于位反转，它们也可从 TI 公司的网站上得到。FFT 函数 cfftr2_dit.sa 假定输入数据 x 为自然顺序，而 FFT 系数或旋转常数是反序的，因此，辅助函数 digitrev_index.c 用来产生位反转的索引指针值，bitrev.sa 实现旋转常数的位反转。在 FFT 函数调用之前，先调用这两个函数。这两个位反转的辅助文件再被调用对扰乱的输出进行位反转。

N 是复输入或输出数据的个数（注意输入数据包括 $2N$ 个元素），以便进行 N 点的 FFT 运算。位反转辅助函数使用函数 SQRT_N。FREQ 通过在数据表中选择每个周期的点数来决定输入正弦数据的频率。当 FREQ 设置为 8 时，则从表中第一个数据点开始，每隔 7 个数据点选一个数据。模运算只用做重新初始化索引指针的标志。下面 4 个点是选择的一个周期：0, 1000, 0 和−1000。例 2.4 演示了用这种指针索引方案选择表中不同的数据点的方法。

利用程序输出 FFT 的幅度谱，下面一行程序是输出语句：

```
output_sample(32000);
```

执行该语句将输出一个幅度接近−1.1 V 的负尖峰脉冲（不是正脉冲，这是因为 AD535 编解码器使用 2 的补码格式和接近 1.1 V 的直流偏置），抽样速率通过轮询方法实现。

建立并运行工程 FFTsinetable，两个用于位反转的辅助文件和复 FFT 函数也需要添加到源工程文件中。图 6.16 显示了结果的时域图（通过 HP 动态信号分析仪获得）。因为每个 T_s 时间进行一次输出，32 点时间间隔就是 $32T_s$，或者是 $32\times(0.125\text{ ms}) = 4\text{ ms}$，每 4 ms 就重复一个负尖峰。这就提供了一个时间参考，因为两个负尖峰的时间间隔对应着 8 kHz 的抽样频率，时间间隔的中间对应着 4 kHz 的奈奎斯特频率（距离负尖峰 2 ms）。第一个正尖峰出现在离第一个负尖峰的 1 ms 处，这对应着频率 $f = F_s/4 = 2\text{kHz}$。第二个正尖峰出现在 3 ms 处，对应折叠频率 $F_s - f = 6\text{ kHz}$。

为了在表中选择 8 个正弦数据，把 FREQ 变为 4。检验输出是否是一个 1 kHz 的信号（类似于用示波器得到的图 6.14）。FREQ 值为 12 时，产生一个 3 kHz 的输出信号；FREQ 为 15 时，在中间出现两个正尖峰（在两个负尖峰间）。注意当频率大于 4 kHz 时，将产生混叠现象。为了说明这一点，把 FREQ 改为 20，检验输出是否是一个在 3 kHz 产生混叠的信号，而不是 5 kHz 的信号。如果把 FREQ 改为 24，输出将是 2 kHz 的混叠信号，而不是 6 kHz 的信号。

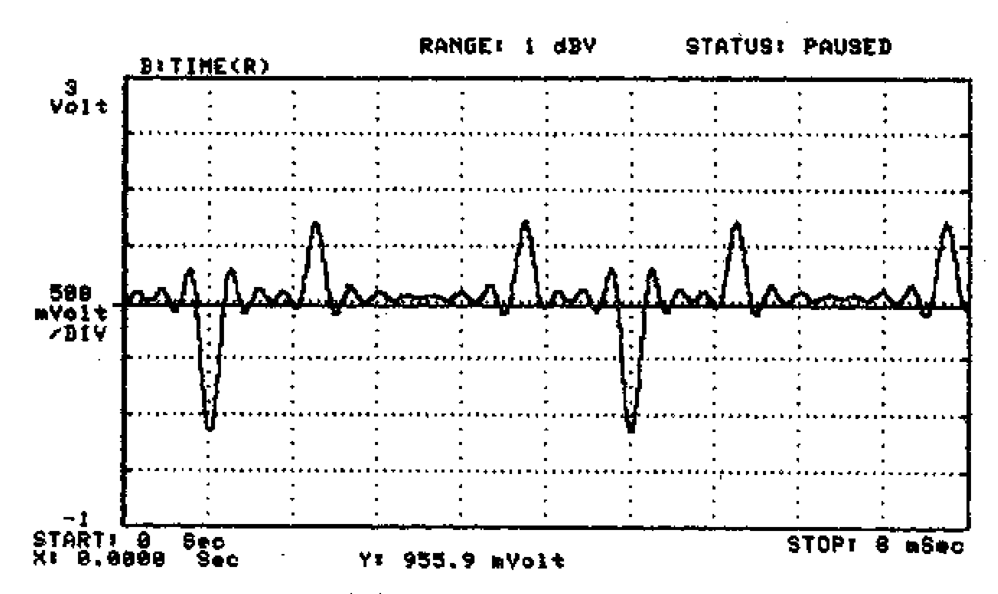


图 6.16 幅度谱的时域图。用表格数据得到 2 kHz 输入数据，利用 TI 公司的 FFT 函数求出 FFT 幅度谱，再画出其时域图

在函数 cfftr2_dit.sa（由 TI 公司提供）中，运行周期数是这样计算的：

```
Cycles = ((2N) + 23)log2(N) + 6
```

对于 1024 点的 FFT，运行周期数为 $2071\times10+6=20\,716$，对应的时间是 $t=20\,716$ 个时钟周期/150 MHz =138 μs。

6.8.1　快速卷积

下面的例子将说明 FFT 在频域内是如何对信号进行处理的，快速卷积[19,20]只需要较少的计算量，且比在时域内实现一个多系数 FIR 滤波器可能更准确。

例 6.4　利用 TI 公司的浮点 FFT 函数进行重叠相加的快速卷积运算，实现 FIR 滤波

图 6.17 给出了 FIR 滤波器的程序 fastconvo.c，并说明了重叠相加的快速卷积算法[19,20]。TI 浮点 FFT 支持函数有 bitrev.sa、digitrev_index.c 和 cfftr2_dit.sa，这些函数已在例 6.3 中介绍过。此外，这里还使用了复 FFT 逆变换函数 icfftr2_dif.sa（频率抽取 DIF 基 2），该函数要求输入序列顺序是扰乱的或经位反转的，因此复 FFT 函数 cfftr2_dit.sa 的输出序号无需再进行位反转，这样在 FFT 变换之后就不需要反转支持文件 digitrev_index.c 和 bitrev.sa 了。数据（抽样）和滤波器的系数的序号都是位反转的，可以按这种顺序相乘。

建立工程 Fastconvo（使用–o1 编译优化级别选项），从文件 coeffs.h 中读出时域滤波器系数，检验输出是否是一个 2 kHz 的带通滤波器。滤波器系数与在例 4.4 中介绍的 BP55.cof 相同，中心频率是 $F_s/4$。

系数文件 coeffs.h 的系数也和 LP55.cof 的系数相同，它表示截止频率为 $F_s/8$的低通滤波器，例 4.4 中也介绍过该系数文件。编辑文件 coeffs.h，实现并检验该低通滤波器。

在该例中，使用了几个缓冲区，iobuffer 是基本的输入输出缓冲区。在每个抽样间隔，执行中断服务程序（ISR）。从 iobuffer 中读出下一个时刻的输出数据，再送到编解码器，然后再输入一个新抽样。经过 PTS/2 个的抽样时间间隔，iobuffer 内有一帧新的 PTS/2 个输入抽样，这时 flag 被设置为 1。

主程序使用 while 循环语句，等待 flag 的变化：

```
while (flag == 0);
```

然后执行下面的操作：

1. 把标志 flag 复位为 0。
2. 把缓冲区 iobuffer 的内容（新输入抽样帧）复制到缓冲抽样存储器的第一个 PTS/2 个缓冲单元。
3. 把缓冲区 overlap 的内容（前面经过计算的输出帧）复制到缓冲区 iobuffer。
4. 处理新的输入抽样帧，算出下一个输出抽样帧。

帧处理周期（在无限循环内）是 PTS/2 个抽样周期，期间执行和包含以下过程：

1. 抽样缓冲区 samples（实部）的最后 PTS/2 个单元内容复制到缓冲区 overlap 中，这些时域数据可认为是前面处理帧的后半部分的重叠（PTS/2 个抽样）。
2. 抽样缓冲区的最后 PTS/2 个存储单元全部写入 0，这样抽样缓冲区内就有 PTS/2 个新样本，后面跟着 PTS/2 个的补零数据。

3. 对抽样缓冲区内的数据进行 PTS 个点的 FFT 变换，将时域抽样变成频域数据。
4. 经 FFT 变换后的频域数据（复数）再乘以存在文件 h 中的复滤波器系数。
5. 再利用 PTS 个点的 IFFT 变换将抽样缓冲区中的频域数据逆变换成时域数据，结果 PTS 个时域数据将是实数。
6. 抽样缓冲区前面 PTS/2 个单元的内容（也就是目前帧处理结果的前半部分）加到重叠缓冲区的内容上去。

```
//FastConvo.c FIR filter implemented using overlap-add fast convolution

#include <math.h>
#include "coeffs.h"                   //time domain FIR coefficients
#define PI 3.14159265358979
#define PTS 256                       //number of points for FFT
#define SQRT_PTS 16                   //used in twiddle factor calc.
#define RADIX 2                       //passed to TI FFT routines
#define DELTA (2*PI)/PTS
typedef struct Complex_tag {float real, imag;} COMPLEX ;
#pragma DATA_ALIGN(W, sizeof(COMPLEX))
#pragma DATA_ALIGN(samples, sizeof(COMPLEX))
#pragma DATA_ALIGN(h, sizeof(COMPLEX))
COMPLEX W[PTS/RADIX] ;                //twiddle factor array
COMPLEX samples[PTS];                 //processing buffer
COMPLEX h[PTS];                       //FIR filter coefficients
short buffercount = 0;                //buffer count for iobuffer samples
float iobuffer[PTS/2];                //primary input/output buffer
float overlap[PTS/2];                 //intermediate result buffer
short i;                              //index variable
short flag = 0;                       //set to indicate iobuffer full
float a, b;                           //variables used in complex multiply
short NUMCOEFFS = sizeof(coeffs)/sizeof(float);
short iTwid[SQRT_PTS] ;               //PTS/2 + 1 > sqrt(PTS)

interrupt void c_int11(void)          //ISR
{
 output_sample((int)(iobuffer[buffercount]));
 iobuffer[buffercount++] = (float)(input_sample());
 if (buffercount >= PTS/2)            //for overlap-add method iobuffer
  {                                   //is half size of FFT used
   buffercount = 0;
   flag = 1;
  }
}

main()
{                                     //set up array of twiddle factors
 digitrev_index(iTwid, PTS/RADIX, RADIX);
 for(i = 0 ; i < PTS/RADIX ; i++)
 {
   W[i].real = cos(DELTA*i);
   W[i].imag = sin(DELTA*i);
```

```
  }
 bitrev(W, iTwid, PTS/RADIX);      //bit reverse order W
 for (i = 0 ; i<PTS ; i++)         //initialise PTS element
  {                                //of COMPLEX to hold real-valued
   h[i].real = 0.0;                //time domain FIR filter coefficients
   h[i].imag = 0.0;
   }
   for (i = 0 ; i < NUMCOEFFS ; i++)
   {                                //read FIR filter coeffs
    h[i].real = coeffs[i];          //NUMCOEFFS should be less than PTS/2
   }
   cfftr2_dit(h,W,PTS);             //transform filter coeffs
   comm_intr();                     //initialise DSK, codec, McBSP
   while(1)                         //frame processing infinite loop
    {
     while (flag == 0);             //wait for iobuffer full
      flag = 0;
      for (i = 0 ; i<PTS/2 ; i++)//iobuffer into first half of
     {                              //samples buffer
      samples[i].real = iobuffer[i];
      iobuffer[i] = overlap[i];    //previously processed output
     }                              //to iobuffer
    for (i = 0 ; i<PTS/2 ; i++)
    {                               //second half of samples to overlap
      overlap[i] = samples[i+PTS/2].real;
     samples[i+PTS/2].real = 0.0; //zero-pad input from iobuffer
    }
    for (i=0; i<PTS ; i++)
      samples[i].imag = 0.0;       //init imag parts in samples buffer
    cfftr2_dit(samples,W,PTS);     //complex FFT function from TI

    for (i=0 ; i<PTS ; i++)        //frequency-domain representation
    {                               //complex multiply samples by h
      a = samples[i].real;
      b = samples[i].imag;
      samples[i].real = h[i].real*a - h[i].imag*b;
      samples[i].imag = h[i].real*b + h[i].imag*a;
    }

    icfftr2_dif(samples,W,PTS);    //inverse FFT function from TI

    for (i=0 ; i<PTS ; i++)
      samples[i].real /= PTS;
    for (i=0 ; i<PTS/2 ; i++)      //add first half of samples
      overlap[i] += samples[i].real;    //to overlap
  }                                 //end of while(1)
 }                                  //end of main()
```

图 6.17 使用 TI 浮点 FFT 函数，实现重叠的快速卷积程序（fastconvo.c）

因为输入输出的信号是实信号，同样缓冲区 iobuffer 和 overlap 也是实数缓冲区，但是，这些信号的频域是用复数形式表示的，抽样缓冲区和滤波器系数数组 h 是复数形式的，每个抽样需要两个浮点值（分别表示实部和虚部）。

快速有效的缓冲方法是使用指针，而不是将数据从一个缓冲区复制到另一个缓冲区，但后者比较直接、清晰明了。

仿真的其他程序版本

程序 fastconvosim.c（在辅助材料中）是 fastconvo.c 非实时的版本程序，该程序对事先存入的输入抽样进行处理。在程序中的特定位置设置断点，用户可在重叠相加处理部分的多个阶段单步执行程序，观察每步每个缓冲区的内容。图 6.18 给出了在处理过程的中间阶段（用 CCS 得到），缓冲区内容 iobuffer，h，samples 和 overlap 的典型结果。

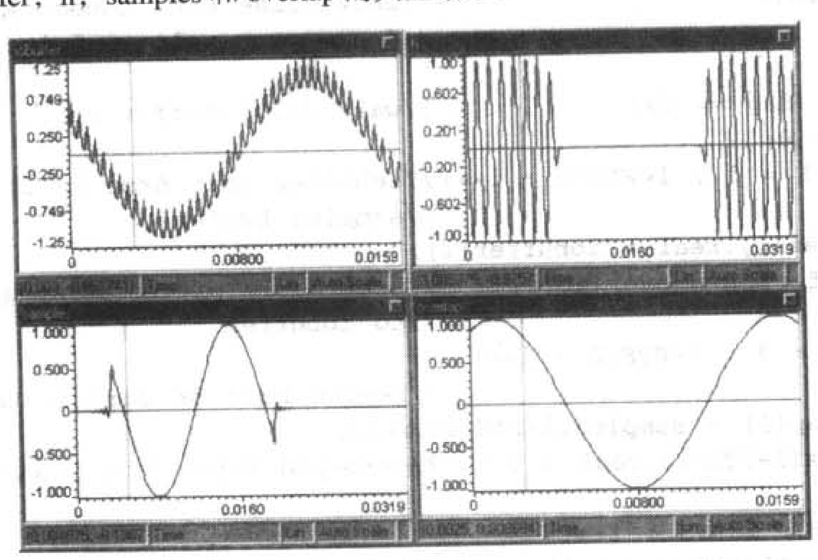


图 6.18　使用仿真程序 fastconvosim.c，在中间处理阶段，利用 CCS 画出 4 个缓冲区 iobuffer，h，samples 和 overlap 的图形

例 6.5　图形均衡器

图 6.19 给出了 graphicEQ.c 的程序清单，该程序实现一个三段图形均衡器。在该工程中，我们再次使用了 TI 浮点复数 FFT 和 IFFT 支持函数（见例 6.3 和例 6.4）。

graphicEQcoeff.h 文件内有三组系数：1.3 kHz 的低通、1.3 kHz 到 2.6 kHz 的带通和 2.6 kHz 的高通，它们都是利用 MATLAB 的函数 fir1 来设计的。输入抽样和三组系数都变成频域数据，滤波是基于例 6.4 的重叠相加方法在频域内实现的[19,20]。注意复数乘法$(H)(X)$运算公式，这里 H 表示传输函数，X 表示输入抽样，相乘的结果是：

$$(H_R + jH_I)(X_R + jX_I) = (H_RX_R - H_IX_I) + j\,(H_RX_I + H_IX_R)$$

程序中同样利用该公式，其中 $j = \sqrt{-1}$。

中断服务程序 ISR 不断地（每个抽样周期T_s）从缓冲区 iobuffer 中输出一个值，然后输入一个新值，直到缓冲区 iobuffer 满时为止，这时又得到一个新的输入数据帧，初始化 iobuffer 指针，并将标志 flag 置位，主程序等待标志 flag 被置位，然后再将它复位。

```
//GraphicEQ.c Graphic Equalizer using TI floating-point FFT functions

#include <math.h>
#include "GraphicEQcoeff.h"        //time-domain FIR coefficients
#define PI 3.14159265358979
#define PTS 256                    //number of points for FFT
#define SQRT_PTS 16
#define RADIX 2
#define DELTA (2*PI)/PTS
typedef struct Complex_tag {float real,imag;} COMPLEX;
#pragma DATA_ALIGN(W,sizeof(COMPLEX))
#pragma DATA_ALIGN(samples,sizeof(COMPLEX))
#pragma DATA_ALIGN(h,sizeof(COMPLEX))
COMPLEX W[PTS/RADIX] ;             //twiddle array
COMPLEX samples[PTS];
COMPLEX h[PTS];
COMPLEX bass[PTS], mid[PTS], treble[PTS];
short buffercount = 0;             //buffer count for iobuffer samples
float iobuffer[PTS/2];             //primary input/output buffer
float overlap[PTS/2];              //intermediate result buffer
short i;                           //index variable
short flag = 0;                    //set to indicate iobuffer full
float a, b;                        //variables for complex multiply
short NUMCOEFFS = sizeof(lpcoeff)/sizeof(float);
short iTwid[SQRT_PTS] ;            //PTS/2+1 > sqrt(PTS)
float bass_gain = 1.0;             //initial gain values
float mid_gain = 0.0;              //change with GraphicEQ.gel
float treble_gain = 1.0;

interrupt void c_int11(void)       //ISR
{
 output_sample((int)(iobuffer[buffercount]));
 iobuffer[buffercount++] = (float)(input_sample());
 if (buffercount >= PTS/2)         //for overlap-add method iobuffer
  {                                //is half size of FFT used
   buffercount = 0;
   flag = 1;
  }
}

main()
{
 digitrev_index(iTwid, PTS/RADIX, RADIX);
 for( i = 0; i < PTS/RADIX; i++ )
  {
   W[i].real = cos(DELTA*i);
   W[i].imag = sin(DELTA*i);
  }
 bitrev(W, iTwid, PTS/RADIX);      //bit reverse W

 for (i=0 ; i<PTS ; i++)
  {
   bass[i].real = 0.0;
   bass[i].imag = 0.0;
   mid[i].real = 0.0;
   mid[i].imag = 0.0;
   treble[i].real = 0.0;
   treble[i].imag = 0.0;
  }
for (i=0; i<NUMCOEFFS; i++)        //same # of coeff for each filter
 {
```

```
    bass[i].real = lpcoeff[i];        //lowpass coeff
    mid[i].real = bpcoeff[i];         //bandpass coeff
    treble[i].real = hpcoeff[i];      //highpass coef
   }

  cfftr2_dit(bass,W,PTS);             //transform each band into frequency
  cfftr2_dit(mid,W,PTS);
  cfftr2_dit(treble,W,PTS);

  comm_intr();                        //initialise DSK, codec, McBSP
  while(1)                            //frame processing infinite loop
   {
    while (flag == 0);                //wait for iobuffer full
           flag = 0;
    for (i=0 ; i<PTS/2 ; i++)         //iobuffer into samples buffer
     {
      samples[i].real = iobuffer[i];
      iobuffer[i] = overlap[i];       //previously processed output
     }                                //to iobuffer
    for (i=0 ; i<PTS/2 ; i++)
     {                                //upper-half samples to overlap
      overlap[i] = samples[i+PTS/2].real;
      samples[i+PTS/2].real = 0.0;    //zero-pad input from iobuffer
     }
    for (i=0 ; i<PTS ; i++)
      samples[i].imag = 0.0;          //init samples buffer

    cfftr2_dit(samples,W,PTS);

    for (i=0 ; i<PTS ; i++)           //construct freq domain filter
     {                                //sum of bass,mid,treble coeffs
     h[i].real = bass[i].real*bass_gain + mid[i].real*mid_gain
               + treble[i].real*treble_gain;
     h[i].imag = bass[i].imag*bass_gain + mid[i].imag*mid_gain
               + treble[i].imag*treble_gain;
     }
    for (i=0; i<PTS; i++)             //frequency-domain representation
     {                                //complex multiply samples by h
      a = samples[i].real;
      b = samples[i].imag;
      samples[i].real = h[i].real*a - h[i].imag*b;
      samples[i].imag = h[i].real*b + h[i].imag*a;
     }

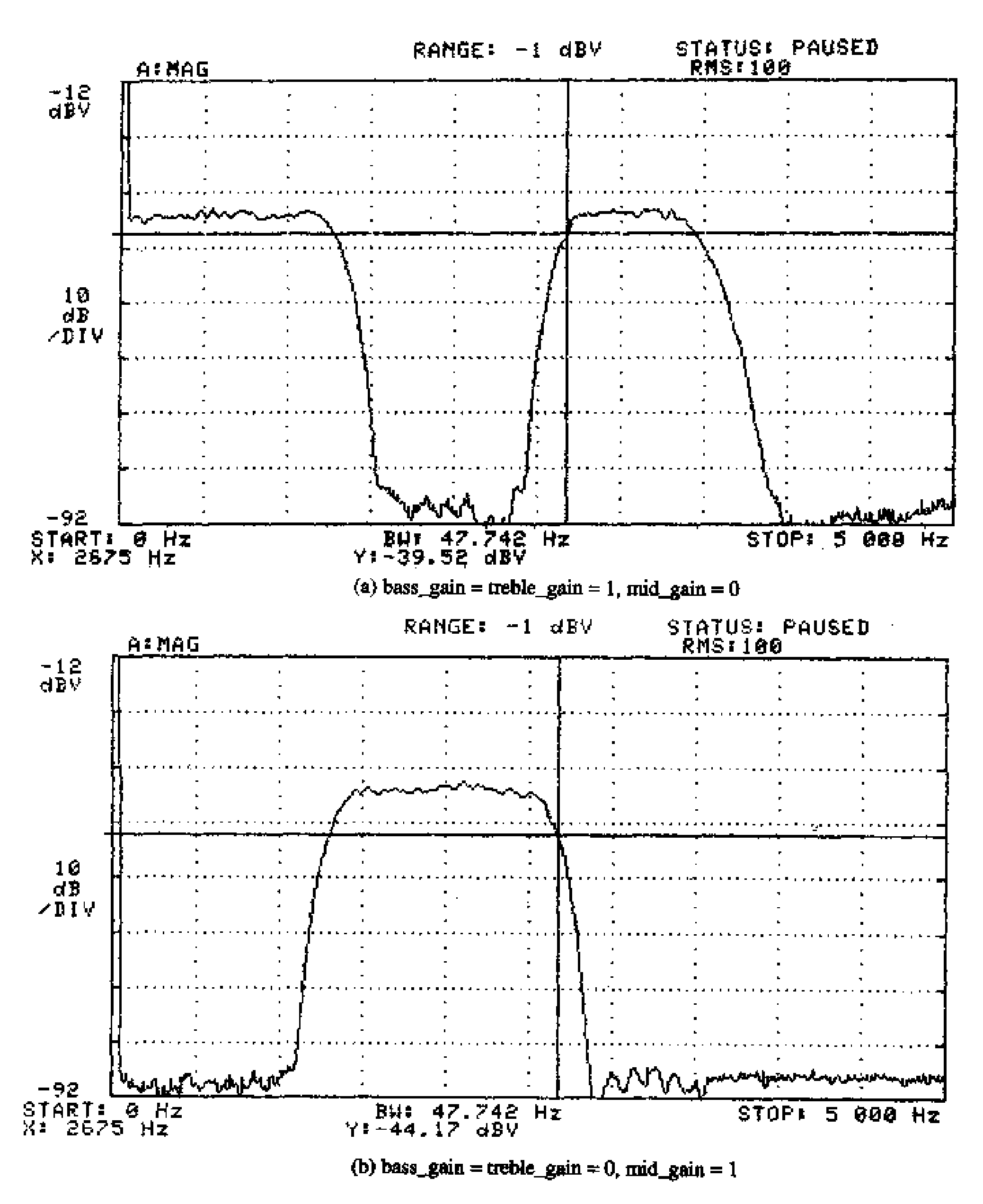
    icfftr2_dif(samples,W,PTS);

    for (i=0 ; i<PTS ; i++)
       samples[i].real /= PTS;
    for (i=0 ; i<PTS/2 ; i++)         //add 1st half to overlap
       overlap[i] += samples[i].real;
   }                                  //end of infinite loop
}                                     //end of main()
```

图 6.19 使用 TI 浮点 FFT 函数的均衡器程序（graphicEQ.c）

创建工程 graphicEQ（使用–o1 优化级选项），使用语音文件，如 TheForce.wav（参见例 4.9）或噪声，测试工程的正确性。检验低频和高频分量被提升了，而中频范围频率分量衰减了，这是

因为程序里滤波器系数定标的缘故，bass_ gain 和 treble_gain 的初始值设为 1，而 mid_gain 的初始值设为 0。滑动条文件 graphicEQ.gel（在辅助材料中）能够独立控制三个频带。图 6.20 给出了利用噪声作为输入和两个不同的增益设置，通过信号分析仪获得的输出谱图形。



(a) bass_gain = treble_gain = 1, mid_gain = 0

(b) bass_gain = treble_gain = 0, mid_gain = 1

图 6.20 频谱分析仪测得的图形均衡器输出功率谱

参考文献

1. J. W. Cooley and J. W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation*, Vol. 19, 1965, pp. 297–301.
2. J. W. Cooley, How the FFT gained acceptance, *IEEE Signal Processing*, Jan. 1992, pp. 10–13.
3. J. W. Cooley, The structure of FFT and convolution algorithms, from a tutorial, *IEEE 1990 International Conference on Acoustics, Speech, and Signal Processing*, Apr. 1990.

4. C. S. Burrus and T. W. Parks, *DFT/FFT and Convolution Algorithms: Theory and Implementation*, Wiley, New York, 1988.
5. G. D. Bergland, A guided tour of the fast Fourier transform, *IEEE Spectrum*, Vol. 6, 1969, pp. 41–51.
6. E. O. Brigham, *The Fast Fourier Transform*, Prentice Hall, Upper Saddle River, NJ, 1974.
7. S. Winograd, On computing the discrete Fourier transform, *Mathematics of Computation*, Vol. 32, 1978, pp. 175–199.
8. H. F. Silverman, An introduction to programming the Winograd Fourier transform algorithm (WFTA), *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-25, Apr. 1977, pp. 152–165.
9. P. E. Papamichalis, ed., *Digital Signal Processing Applications with the TMS320 Family: Theory, Algorithms, and Implementations*, Vol. 3, Texas Instruments, Dallas, TX, 1990.
10. R. N. Bracewell, Assessing the Hartley transform, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-38, 1990, pp. 2174–2176.
11. R. N. Bracewell, *The Hartley Transform*, Oxford University Press, New York, 1986.
12. H. V. Sorensen, D. L. Jones, M. T. Heidman, and C. S. Burrus, Real-valued fast Fourier transform algorithms, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-35, 1987, pp. 849–863.
13. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, Wiley, New York, 1999.
14. R. Chassaing, *Digital Signal Processing with C and the TMS320C30*, Wiley, New York, 1992.
15. P. M. Embree and B. Kimble, *C Language Algorithms for Digital Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1990.
16. S. Kay and R. Sudhaker, A zero crossing spectrum analyzer, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34, Feb. 1986, pp. 96–104.
17. P. Kraniauskas, A plain man's guide to the FFT, *IEEE Signal Processing*, Apr. 1994.
18. J. R. Deller, Jr., Tom, Dick, and Mary discover the DFT, *IEEE Signal Processing*, Apr. 1994.
19. A. V. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1989.
20. J. G. Proakis and D. G. Manolakis, *Digital Signal Processing*, Upper Saddle River, NJ, 1996.

第 7 章　自适应滤波器

本章的主要内容包括：（1）自适应滤波器的结构；（2）最小均方（LMS）算法；（3）噪声抵销和系统辨识应用的 C 语言程序实例。

自适应滤波器最适用于信号或系统参数缓慢变化的情况，这时滤波器自动调整参数，以补偿这些参数变化造成的影响。最小均方（LMS）准则是一种可用于调整滤波器系数的搜索算法。本章同时给出了一些程序实例，提供了对自适应滤波器基本的、直观的理解。

7.1　引言

对于常规的 FIR 和 IIR 数字滤波器，决定滤波器特性的过程参数是已知的，尽管它们可能随着时间变化，但是这种变化特性是先验已知的。在实际应用中，由于没有足够的关于过程的先验数据，因此很多参数就存在很大的不确定性，有些系数甚至可能会随时间变化，而且确切的变化特性是很难预测出来的。在这些情况下，人们非常希望能够设计出随着当时条件的变化，自动地调整参数的自学习滤波器。

自适应滤波器的系数能自动调节，以补偿输入信号、输出信号或系统参数的变化。自适应系统能了解信号的特性，跟踪缓慢变化的信号，而不是固定不变的。在未知信号特性或信号特性随着时间变化的场合，自适应滤波器是非常用的。

图 7.1 表示的是自适应滤波器的基本结构，其输出 y 与所期望的信号 d 相比较产生一个误差信号 e，该误差信号反馈回自适应滤波器。基于误差信号，运用最小均方（LMS）的算法，调整或优化滤波器的系数。

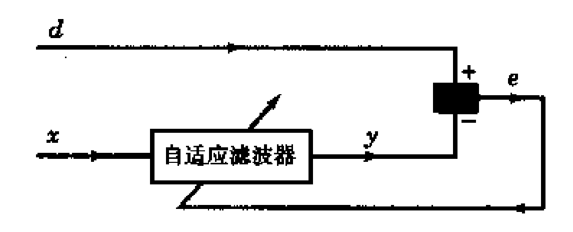


图 7.1　自适应滤波器的基本结构

尽管目前有许多方法来实现自适应滤波器，但这里我们只讨论结合线性合并（FIR 滤波器）的 LMS 搜索算法。图 7.1 自适应滤波器的输出为：

$$y(n)=\sum_{k=0}^{N-1} w_k(n)x(n-k) \tag{7.1}$$

其中 $w_k(n)$ 表示特定时刻 n 的 N 个权重或系数。在第 4 章中，式 7.1 的卷积方程是结合 FIR 滤波器来实现的。在自适应滤波和神经网络中用术语权重 w 代替系数是通常的做法。

对于一个滤波器，需要一个性能标准来决定滤波器的好坏，通常性能标准是基于误差函数的：

$$e(n)=d(n)-y(n) \tag{7.2}$$

误差函数是期望信号 $d(n)$ 与自适应滤波器输出 $y(n)$ 的差值。调节权重或系数 $w_k(n)$，使得误差的均方值最小。均方误差函数是指 $E[e^2(n)]$，E 表示期望值。因为有 k 个权重或系数，需要求均方误差函数的导数。可以找到一种估计方法来替代对 $e^2(n)$ 求导，从而得到：

$$w_k(n+1)=w_k(n)+2\beta e(n)x(n-k) \quad k=0,1,\cdots,N-1 \tag{7.3}$$

上式就是LMS算法[1-3]。式（7.3）提供了一种不需要计算均值或求微分，只需调整更新权重或系数，从而实现自适应滤波器的简单且强有效的方法。$x(n)$是自适应滤波器的输入，β表示收敛速度和自适应过程的准确度（自适应步长）。

对于每个特定的时刻 n，如果误差信号 $e(n)$ 不为0，就根据式（7.3）替代或更新权重或系数。对于一个特定时刻 n，滤波器输出 $y(n)$ 后，误差信号 $e(n)$、权重或系数 $w_k(n)$ 又被更新，ADC又获得一个新抽样数据，这种自适应过程在不同时间不断重复执行。注意式（7.3）中，当误差信号 $e(n)$ 为0时，权重或系数是无需调整的。

线性自适应合并是一种最有用的自适应滤波器结构，是一种可调的FIR滤波器。但第4章中讨论的频率选择性FIR滤波器的系数是固定的，自适应滤波器的系数或权重可随环境的变化而调整，如输入信号的变化。为了满足特定任务的要求，也可以使用自适应IIR滤波器（这里不做讨论）。对于自适应IIR滤波器来说，最主要的问题是在自适应变化过程中，它的极点很可能处于单位圆外，从而导致滤波器的不稳定。

下面将使用式（7.1）~（7.3）编写开发相应的程序，在式（7.3）中，只用变量β而不用2β。

7.2 自适应滤波器结构

在自适应滤波器设计中，有许多不同的自适应滤波器结构满足不同的应用要求。

1. 噪声抵销。图7.2给出了由图7.1改进的且适用于噪声抵销的自适应滤波器结构。期望信号 d 被不相关的加性噪声 n 所污染，自适应滤波器的输入噪声 n' 与噪声 n 相关，噪声 n' 和 n 可能来自同样的噪声源，但被环境改变了。自适应滤波器的输出 y 适应噪声信号的变化，这时，误差信号接近于期望信号 d，所有的输出信号是误差信号，而不是自适应滤波器的输出信号 y。我们将用C语言编程实例进一步说明这种滤波器结构。
2. 系统辨识。图7.3给出了用于系统辨识或建模的自适应滤波器结构。相同的信号输入到未知系统以及与之并联自适应滤波器，误差信号 e 是未知系统输出 d 和自适应滤波器输出 y 的差值，该误差信号反馈到自适应滤波器，用于调整自适应滤波器的系数，直到所有的输出 $y=d$。这时自适应过程结束，而 e 趋于0。在该方案中，自适应滤波器就构造该未知滤波器的模型，后面将通过三个编程实例来说明这种结构。

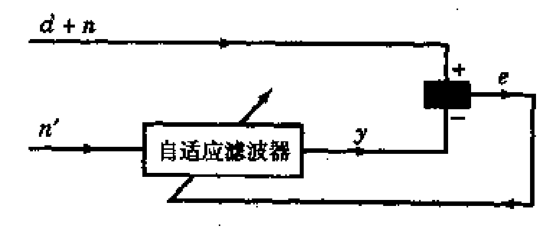


图7.2　噪声抵销自适应滤波器结构

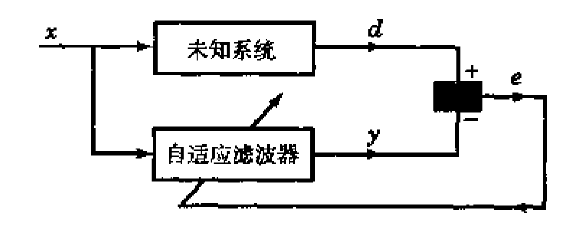


图 7.3　系统辨识滤波器结构

3. 自适应预测器。图 7.4 给出了能估计输入信号的自适应预测器结构，后面将通过一个编程例子来说明它。
4. 其他结构，如：
   (a) 具有两个权重的陷波器。用于消除或衰减一个正弦噪声信号，这种结构只有两个权重系数，图 7.5 给出了它的结构，文献[1,3,4]用 C31 处理器说明了这种结构。
   (b) 自适应信道均衡。应用于调制解调器中，用来减小在电话信道中传输高速数据引起的失真。

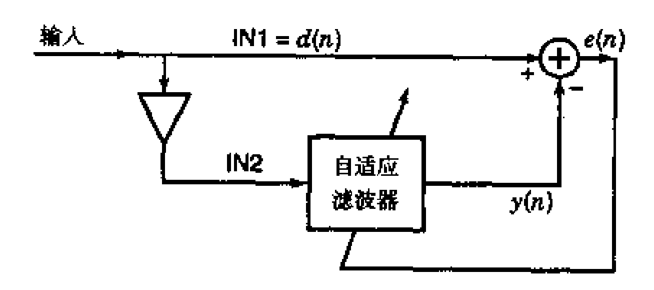


图 7.4　自适应预测器结构

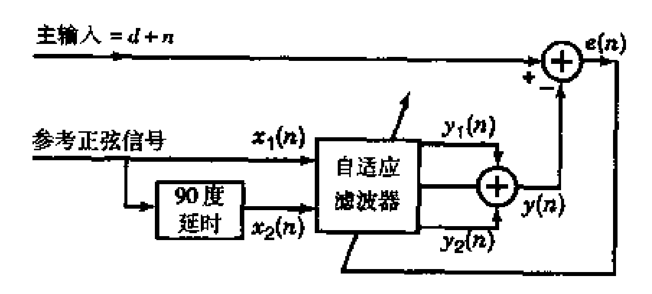


图 7.5　两个权重的自适应陷波器结构

LMS 算法适合于多种应用，包括自适应回波和噪声抵销，自适应均衡和自适应预测等。

基于 LMS 算法，已得到许多演变的算法，例如符号-误差 LMS、符号-数据 LMS 和符号-符号 LMS 算法等。

1. 对于符号-误差 LMS 算法，式（7.3）变为：

$$w_k(n+1)=w_k(n)+\beta\,\mathrm{sgn}[e(n)]x(n-k) \tag{7.4}$$

其中 sgn 为符号函数：

$$\mathrm{sgn}(u)=\begin{cases}1, & \text{如果 } u\geqslant 0\\ -1, & \text{如果 } u<0\end{cases} \tag{7.5}$$

2. 对于符号-数据 LMS 算法，式（7.3）变为：

$$w_k(n+1)=w_k(n)+\beta e(n)\,\mathrm{sgn}[x(n-k)] \tag{7.6}$$

3．对于符号-符号 LMS 算法，式（7.3）变为：

$$w_k(n+1)=w_k(n)+\beta\,\mathrm{sgn}[e(n)]\,\mathrm{sgn}[x(n-k)] \tag{7.7}$$

可化简为：

$$w_k(n+1)=\begin{cases}w_k(n)+\beta & \mathrm{sgn}[e(n)]=\mathrm{sgn}[x(n-k)]\\ w_k(n)-\beta & 其他\end{cases} \tag{7.8}$$

从数学角度来看，公式变得更简单了，因为该算法没有乘法运算。

这些演变的算法没有利用 TMS320C6x 处理器的流水线特性。由于包含涉及误差信号或抽样数据符号测试的判决类型指令，TMS320C6x 处理器运行这些演变的算法速度可能比基本 LMS 算法慢。

LMS 算法非常适合应用于自适应均衡器、电话的回音抵销器等设备中。其他算法，如递归最小二乘（RLS）算法[4]，比基本 LMS 收敛快，但需要较大的计算量。RLS 基于最优化方案，用每个输入样本来更新冲激响应以获得最佳性能，在输入每个时间抽样进行计算时，都需定义正确的步长和方向。

文献[4]中介绍了用于恢复信号特性的自适应算法。当不能获得合适的参考信号时，这些算法是非常有用的。在实现自适应滤波之前，滤波器会自适应调整，以恢复信号丢失的特性。和在 LMS 和 RLS 算法中一样，不是利用期望的波形作为模板，这种特性用于滤波器的自适应调整。当可以利用期望的信号时，可使用如 LMS 的常规算法；否则，就需要信号的先验知识了。

7.3　噪声抵销和系统辨识的编程实例

该例介绍了使用最小均方（LMS）算法实现自适应滤波的过程，即使该例没有使用 DSK，阅读它也是有指导意义的，这是因为它说明了自适应处理的步骤。

例 7.1　用 Borland C/C++编译 C 语言程序实现自适应滤波器

该 Borland C/C++编译的 C 语言程序例子采用了 LMS 算法，它说明了利用图 7.1 的自适应结构实现自适应滤波的过程：

1．获得一个新抽样、期望信号 d 以及自适应滤波器 x 的参考输入噪声信号。
2．与第 4 章中的 FIR 滤波器一样，利用式（7.1）计算自适应 FIR 滤波器的输出 y。在图 7.1 的结构中，所有输出与自适应滤波器的输出 y 相同。
3．利用式（7.2）计算误差信号。
4．利用式（7.3）更新系数或权重。
5．利用第 4 章中的数据移动方案，更新下个时刻 n 的输入抽样数据。这种方案移动数据，而不是移动指针。
6．对于下一个输出抽样点，重复整个自适应过程。

图 7.6 给出了基于 LMS 算法实现图 7.1 自适应滤波器结构的程序 adaptc.c。选 $2\cos(2n\pi f/F_s)$ 作为期望信号，$\sin(2n\pi f/F_s)$ 作为自适应滤波器的参考噪声输入，其中 f 为 1 kHz，F_s = 8 kHz。自适应速率、滤波器阶数和抽样点数分别是 0.01, 22 和 40。

所有输出是自适应滤波器的输出 y，该信号自适应或收敛于期望的余弦信号 d。

```
//Adaptc.c Adaptation using LMS without TI's compiler

#include <stdio.h>
#include <math.h>
#define beta 0.01                          //convergence rate
#define N 21                               //order of filter
#define NS 40                              //number of samples
#define Fs 8000                            //sampling frequency
#define pi 3.1415926
#define DESIRED 2*cos(2*pi*T*1000/Fs)      //desired signal
#define NOISE sin(2*pi*T*1000/Fs)          //noise signal

main()
{
 long I, T;
 double D, Y, E;
 double W[N+1] = {0.0};
 double X[N+1] = {0.0};
 FILE *desired, *Y_out, *error;
 desired = fopen ("DESIRED", "w++");   //file for desired samples
 Y_out = fopen ("Y_OUT", "w++");       //file for output samples
 error = fopen ("ERROR", "w++");       //file for error samples
 for (T = 0; T < NS; T++)              //start adaptive algorithm
  {
  X[0] = NOISE;                        //new noise sample
  D = DESIRED;                         //desired signal
  Y = 0;                               //filter'output set to zero
  for (I = 0; I <= N; I++)
   Y += (W[I] * X[I]);                 //calculate filter output
  E = D - Y                            //calculate error signal
  for (I = N; I >= 0; I--)
   {
    W[I] = W[I] + (beta*E*X[I]);       //update filter coefficients
    if (I != 0)
    X[I] = X[I-1];                     //update data sample
   }
  fprintf (desired, "\n%10g  %10f", (float) T/Fs, D);
  fprintf (Y_out, "\n%10g  %10f", (float) T/Fs, Y);
  fprintf (error, "\n%10g  %10f", (float) T/Fs, E);
  }
  fclose (desired);
  fclose (Y_out);
  fclose (error);
}
```

图 7.6　Borland C/C++编译的自适应滤波器程序（adaptc.c）

源文件用 Borland C/C++编译器进行编译。执行该程序，图 7.7 画出了自适应滤波器输出（y_out）收敛于期望余弦信号的图形，将自适应或收敛速率β改为 0.02，检验自适应过程的速率是否提高了。

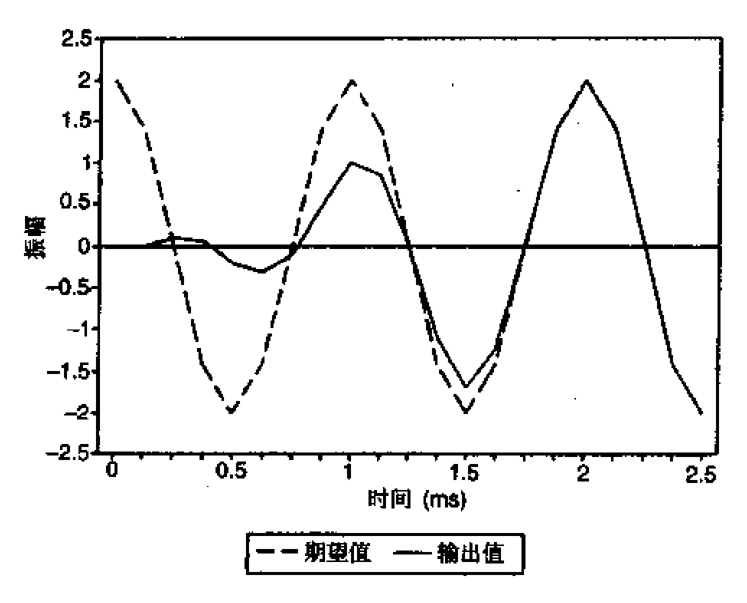


图 7.7 自适应滤波器输出收敛于期望余弦信号的图形

自适应交互过程

图 7.6 中，程序 adaptc.c 的另一个版本是辅助材料中的 adaptive.c，它使用 Turbo 或 Borland C/C++编译，具有图形和交互能力，能画出随β的不同自适应变化的过程。它使用幅度为 1 的期望余弦信号和 31 阶的滤波器。执行程序，输入β为 0.01，检验图 7.8 的结果。注意输出收敛于期望的余弦信号。按下 F2 键，再次运行程序，输入不同的β值，检验输出结果。

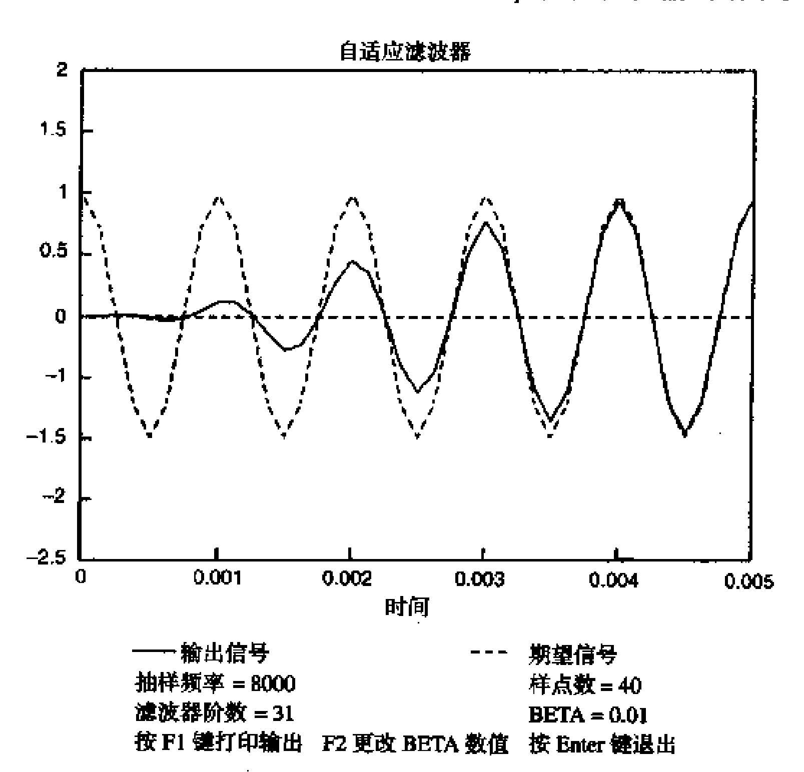


图 7.8 利用程序 adaptive.c 的交互功能，画出的自适应滤波器输出收敛于期望余弦信号的过程

例 7.2 噪声抵销的自适应滤波器

这个例子说明使用 LMS 准则消除不希望的正弦噪声的方法。图 7.9 给出了使用图 7.1 结构实现自适应 FIR 滤波器的程序 adaptnoise.c。该程序使用浮点数据格式，使用整数数据格式的程序 adaptnoise_ int.c 包含在辅助材料中。

1500 Hz 的期望信号以及叠加的 312 Hz 加性（不期望）正弦信号作为自适应滤波器结构的两个输入中的一个输入信号，频率 312 Hz 的参考（模板）余弦信号作为有 30 个系数的自适应 FIR 滤波器输入。312 Hz 的参考信号和 312 Hz 的加性正弦噪声信号进行相关，而不是和期望的 1500 Hz 正弦信号进行相关。

对于每个时刻 n，计算自适应 FIR 滤波器的输出，且 30 个权重或系数随延时抽样的输入一起更新，该“误差”信号 E 是自适应滤波结构期望的全部输出，误差信号是期望信号与加性噪声信号（dplusn）的差值，也是自适应滤波器的输出 $y(n)$。

由查找表产生的所有信号都是用 MATLAB 生成的，该例中没有使用外部输入。图 7.10 给出了计算作为 1500 Hz 期望正弦信号、312 Hz 加性正弦噪声信号以及 312 Hz 的余弦参考信号的 MATLAB 程序 adaptnoise.m（在辅助材料中有一个比较完整的版本），该程序计算了理想的 1500 Hz 的正弦信号、附加 312 Hz 的正弦噪声和 312 Hz 的余弦参考信号的数据值，生成的文件（在辅助材料中）分别是：

1. dplusn：sine(1500 Hz) + sine(312 Hz)
2. refnoise：cosine(312 Hz)

图 7.11 给出了表示 1500 Hz 期望正弦信号的文件 sin1500.h，由文件 sin1500.h 产生的信号频率是：

$$f = F_s\,(\#\text{ of cycles})/(\#\text{ of points}) = 8000(24)/128 = 1500\text{ Hz}$$

常数 beta 决定收敛速率。

```
//Adaptnoise.c Adaptive FIR filter for noise cancellation

#include <refnoise.h>               //cosine 312Hz
#include <dplusn.h>                 //sin(1500) + sin(312)
#define beta 1E-9                   //rate of convergence
#define N 30                        //# of weights (coefficients)
#define NS 128                      //# of output sample points
float w[N];                         //buffer weights of adapt filter
float delay[N];                     //input buffer to adapt filter
short output;                       //overall output
short out_type = 1;                 //output type for slider

interrupt void c_int11()            //ISR
{
 short i;
 static short buffercount=0;        //init count of # out samples
 float yn, E;                       //output filter/"error" signal

 delay[0] = refnoise[buffercount];  //cos(312Hz) input to adapt FIR
 yn = 0;                            //init output of adapt filter

 for (i = 0; i < N; i++)            //to calculate out of adapt FIR
    yn += (w[i] * delay[i]);        //output of adaptive filter
```

```
 E = dplusn[buffercount] - yn;      //"error" signal=(d+n)-yn

 for (i = N-1; i >= 0; i--)         //to update weights and delays
   {
    w[i] = w[i] + beta*E*delay[i]; //update weights
    delay[i] = delay[i-1];          //update delay samples
   }
 buffercount++;                     //increment buffer count
 if (buffercount >= NS)             //if buffercount=# out samples
    buffercount = 0;                //reinit count

 if (out_type == 1)                 //if slider in position 1
    output = ((short)E*10);         //"error" signal overall output
 else if (out_type == 2)
    output=dplusn[buffercount]*10;  //desired(1500)+noise(312)

 output_sample(output);             //overall output result
 return;                            //return from ISR
}

void main()
{
 short T=0;
 for (T = 0; T < 30; T++)
   {
    w[T] = 0;                       //init buffer for weights
    delay[T] = 0;                   //init buffer for delay samples
   }
 comm_intr();                       //init DSK, codec, McBSP
 while(1);                          //infinite loop
}
```

图 7.9 噪声抵销的自适应FIR滤波器程序(adaptnoise.c)

```
%Adaptnoise.m Generates: dplusn.h, refnoise.h, sin1500.h

for i=1:128
  desired(i) = round(100*sin(2*pi*(i-1)*1500/8000)); %sin(1500)
  addnoise(i) = round(100*sin(2*pi*(i-1)*312/8000)); %sin(312)
  refnoise(i) = round(100*cos(2*pi*(i-1)*312/8000)); %cos(312)
end

dplusn = addnoise + desired;                 %sin(312) + sin(1500)

fid=fopen('sin1500.h','w');                  %desired sin(1500)
fprintf(fid,'short sin1500[128]={');
fprintf(fid,'%d, ' ,desired(1:127));
fprintf(fid,'%d' ,desired(128));
fprintf(fid,'};\n');
fclose(fid);

% fid=fopen('dplusn.h','w');                 %desired + noise
% fid=fopen('refnoise.h','w');               %reference noise
```

图 7.10 产生 sine(1500), sine(1500) + sine(312)和 cosine(312)数据的 MATLAB 程序(adaptnoise.m)

```
short sin1500[128]={0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92, 0, 92, 71, -38, -100, -38, 71, 92, 0, -92, -71, 38,
100, 38, -71, -92};
```

图 7.11　生成 128 点 sine(1500 Hz)数据文件的 MATLAB 头文件（sin1500.h）

创建并运行工程 adaptnoise，检验下面的输出结果：期望的 1500 Hz 信号保留下来而不期望的 312 Hz 的正弦信号逐渐减小（被消除）。注意在该应用里，期望的输出是误差信号 E，它自适应（收敛）于期望的信号。当增大 beta 时，消除不期望信号的速率会更快，但 beta 过大，就无法观察到自适应过程，因为这时输出就和 1500 Hz 信号一样。将滑动条设置在位置 2 时，输出（dplusn）就是 312 Hz 噪声信号和期望的 1500 Hz 正弦信号叠加的信号。

例 7.3　用自适应 FIR 滤波器实现固定系数 FIR 滤波器的系统辨识

图 7.12 给出了对未知系统建模或识别未知系统的程序 adaptIDFIR.c。也可参见例 7.2，它用自适应 FIR 滤波器消除噪声。

检验该自适应方案，待识别的未知系统是一个中心频率为 $F_s/4 = 2$ kHz 的 55 阶 FIR 带通滤波器。这个固定系数的 FIR 滤波器系数包含在第 4 章已经介绍过的文件 bp55.cof 中。我们使用一个 60 阶系数自适应 FIR 滤波器模拟这个未知固定系数的 FIR 带通滤波器。

在程序内产生的伪随机噪声序列（见例 2.16 和例 4.4），并把它分别作为未知固定系数滤波器和自适应 FIR 滤波器的输入，输入信号表示训练信号。自适应过程一直继续下去，直到误差信号达到最小。反馈的误差信号就是未知固定系数 FIR 滤波器和自适应 FIR 滤波器输出的差值。

在每个延迟抽样缓冲区（固定系数和自适应 FIR）中，有一个额外的存储单元，它用于更新延时抽样（参见例 4.8 中的 B 方法）。

建立并运行工程 adaptIDFIR（使用 C67x 浮点工具），检验自适应滤波器的输出（adaptfir_out）是一个中心频率为 2 kHz 的带通滤波器（在滑动条选项在默认位置 1）。当滑动条设置在位置 2 时，检验由系数文件 bp55.cof 表示的、中心频率为 2 kHz 的固定系数 FIR 带通滤波器的输出（fir_out），这时能观察到它的输出实际上与自适应滤波器的输出一样。

编辑主函数，添加系数文件 BS55.cof（在例 4.4 中介绍过），它表示一个中心频率为 2 kHz、55 阶系数的 FIR 带阻滤波器，将该带阻滤波器作为待识别的未知系统。

重新建立和运行工程，检验自适应 FIR 滤波器（滑动条设置在位置 1）输出是否和 FIR 带阻滤波器（滑动条设置在位置 2）实际上是一致的。将 beta 以 10 为倍数增大（或减小），观察收敛速度加快（减慢）的情况。将权重（系数）的个数由 60 减小到 40，检验识别过程是否稍微有点变差。

例 7.4　将权重初始化为 FIR 带通滤波器，用自适应滤波器实现固定系数 FIR 系统的识别

将例 7.3 程序 adaptIDFIR.c 稍做改动，建立程序 adaptIDFIRW.c（在辅助材料中），新程序用

中心频率为 3 kHz 的 FIR 带通滤波器系数文件 bp3000.cof（在辅助材料中）初始化自适应 FIR 滤波器的权值。在 main 函数里，权值 $w[i]$不用 0 初始化，而用文件 bp3000.cof 中的系数初始化。

```
//AdaptIDFIR.c Adaptive FIR for system ID of an FIR (uses C67 tools)

#include "bp55.cof"                    //fixed FIR filter coefficients
#include "noise_gen.h"                 //support noise generation file
#define beta 1E-13                     //rate of convergence
#define WLENGTH 60                     //# of coefffor adaptive FIR
float w[WLENGTH+1];                    //buffer coeff for adaptive FIR
int dly_adapt[WLENGTH+1];              //buffer samples of adaptive FIR
int dly_fix[N+1];                      //buffer samples of fixed FIR
short out_type = 1;                    //output for adaptive/fixed FIR
int fb;                                //feedback variable
shift_reg sreg;                        //shift register

int prand(void)                        //pseudo-random sequence {-1,1}
{
 int prnseq;
 if(sreg.bt.b0)
     prnseq = -8000;                   //scaled negative noise level
 else
     prnseq = 8000;                    //scaled positive noise level
 fb =(sreg.bt.b0)^(sreg.bt.b1);        //XOR bits 0,1
 fb^=(sreg.bt.b11)^(sreg.bt.b13);      //with bits 11,13 -> fb
 sreg.regval<<=1;
 sreg.bt.b0=fb;                        //close feedback path
 return prnseq;                        //return noise sequence
}

interrupt void c_int11()               //ISR
{
 int i;
 int fir_out = 0;                      //init output of fixed FIR
 int adaptfir_out = 0;                 //init output of adapt FIR
 float E;                              //error=diff of fixed/adapt out

 dly_fix[0] = prand();                 //input noise to fixed FIR
 dly_adapt[0]=dly_fix[0];              //as well as to adaptive FIR

 for (i = N-1; i>= 0; i--)
  {
   fir_out +=(h[i]*dly_fix[i]);        //fixed FIR filter output
   dly_fix[i+1] = dly_fix[i];          //update samples of fixed FIR
  }

 for (i = 0; i < WLENGTH; i++)
   adaptfir_out +=(w[i]*dly_adapt[i]); //adaptive FIR filter output
```

```
  E = fir_out - adaptfir_out;                  //error signal

  for (i = WLENGTH-1; i >= 0; i--)
   {
    w[i] = w[i]+(beta*E*dly_adapt[i]);         //update weights of adaptive FIR
    dly_adapt[i+1] = dly_adapt[i];             //update samples of adaptive FIR
   }

  if (out_type == 1)                           //slider position for adapt FIR
    output_sample(adaptfir_out);               //output of adaptive FIR filter
  else if (out_type == 2)                      //slider position for fixed FIR
    output_sample(fir_out);                    //output of fixed FIR filter
  return;
 }

void main()
{
  int T=0, i=0;
  for (i = 0; i < WLENGTH; i++)
   {
    w[i] = 0.0;                                //init coeff for adaptive FIR
    dly_adapt[i] = 0;                          //init buffer for adaptive FIR
   }
  for (T = 0; T < N; T++)
    dly_fix[T] = 0;                            //init buffer for fixed FIR

  sreg.regval=0xFFFF;                          //initial seed value
  fb = 1;                                      //initial feevack value
  comm_intr();                                 //init DSK, codec, McBSP
  while (1);                                   //infinite loop
}
```

图 7.12　用自适应滤波器模拟（识别）固定系数 FIR 滤波器的程序（adaptIDFIR.c）

建立工程 adaptIDFIRw（使用 C67x 浮点工具）。最初，自适应 FIR 滤波器输出频谱是中心频率为 3 kHz 的 FIR 带通滤波器，然后，当参考滤波器逐渐停止输出时，输出频谱逐渐调整（收敛）于中心频率为 2 kHz 固定系数（未知）的 FIR 带通滤波器（用 bp55.cof 表示）。当自适应发生时，在有些时间可以观察到两个带通滤波器，为了便于观察，通常希望慢慢地增加自适应的速率（beta 值）。

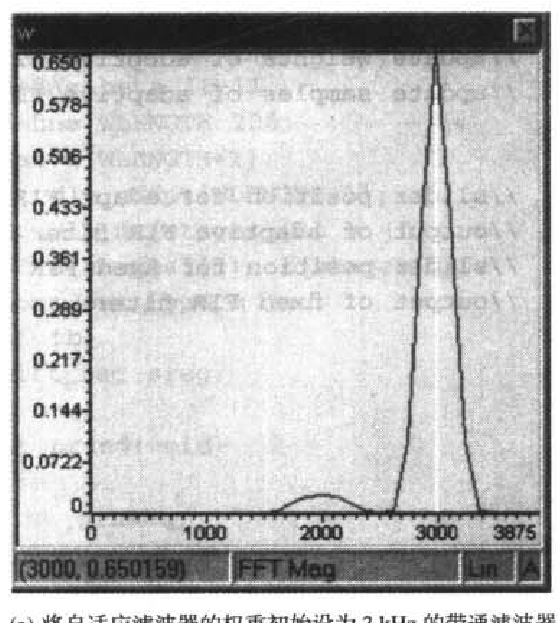
图 7.13 显示了用 CCS 图进行自适应的过程，图 7.14 显示了用 HP 动态信号分析仪进行实时自适应的过程。

例 7.5　自适应 FIR 用于固定系数 IIR 系统的识别

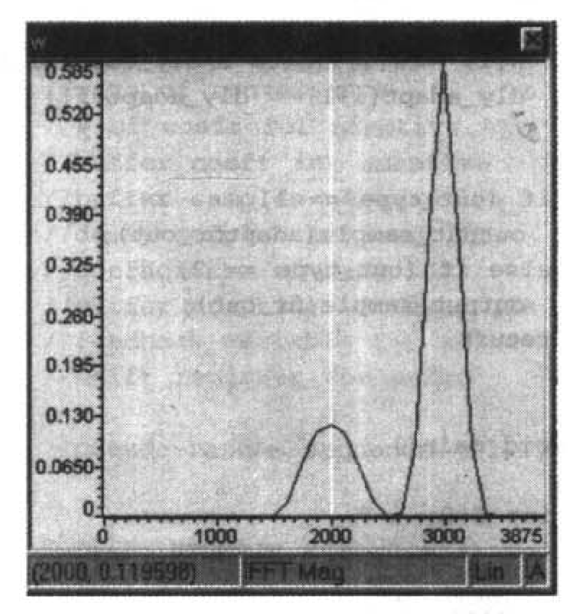
图 7.15 给出了程序 adaptIDIIR.c，它使用自适应 FIR 滤波器模拟或识别（固定系数的未知 IIR）系统。参见实现 IIR 滤波器的例 5.1 以及实现自适应 FIR 滤波器的例 7.3 和例 7.4，后两个例子实现固定系数 FIR 滤波器的建模。

为了测试自适应方案，选择一个未知待识别的带通滤波器，滤波器是中心频率为 2 kHz 的 36 阶 IIR 滤波器，有 18 个二阶单元，滤波器系数包含在例 5.1 介绍的文件 bp2000.cof 中。使用 200

个系数的自适应 FIR 滤波器来构建固定系数未知的 IIR 带通滤波器，但为了建一个更好的模型，需要比该自适应滤波器更多的系数或权值。

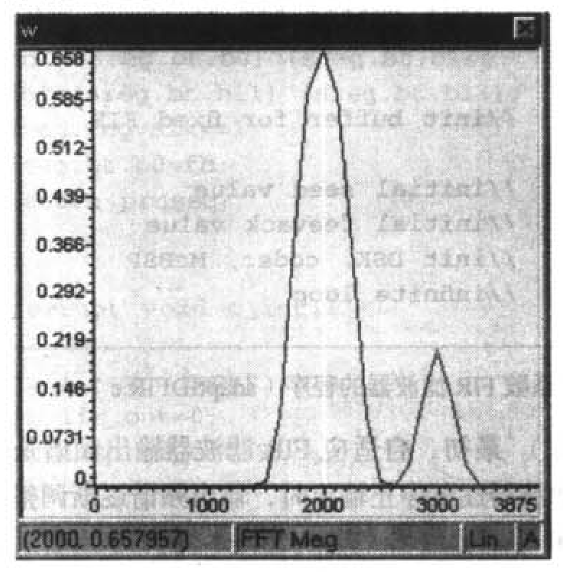


(a) 将自适应滤波器的权重初始设为 3 kHz 的带通滤波器

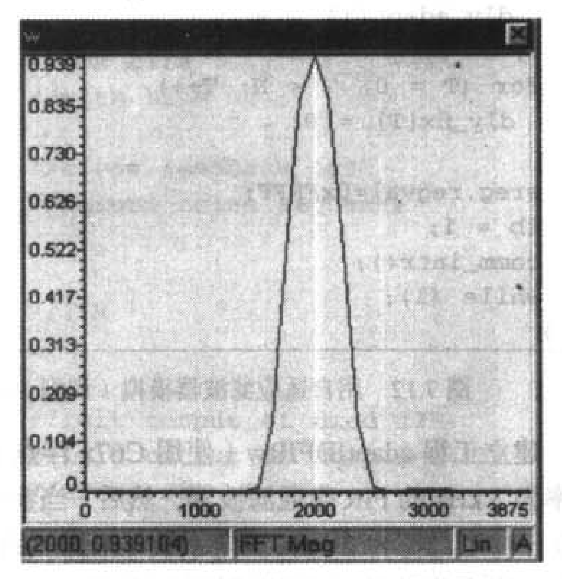




(b) 权重开始收敛于 2 kHz 的滤波器



(c) 当使用简化的 3 Hz 滤波器时，权重收敛于 2 kHz 的滤波器



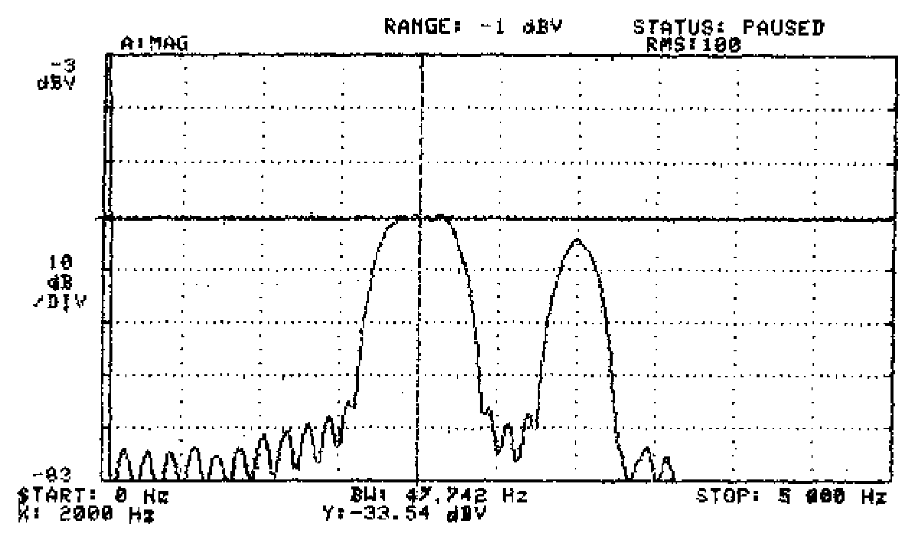
(d) 收敛于 2 kHz 的带通滤波器，自适应过程结束

图 7.13　CCS 画出的显示自适应滤波器自适应过程的图形

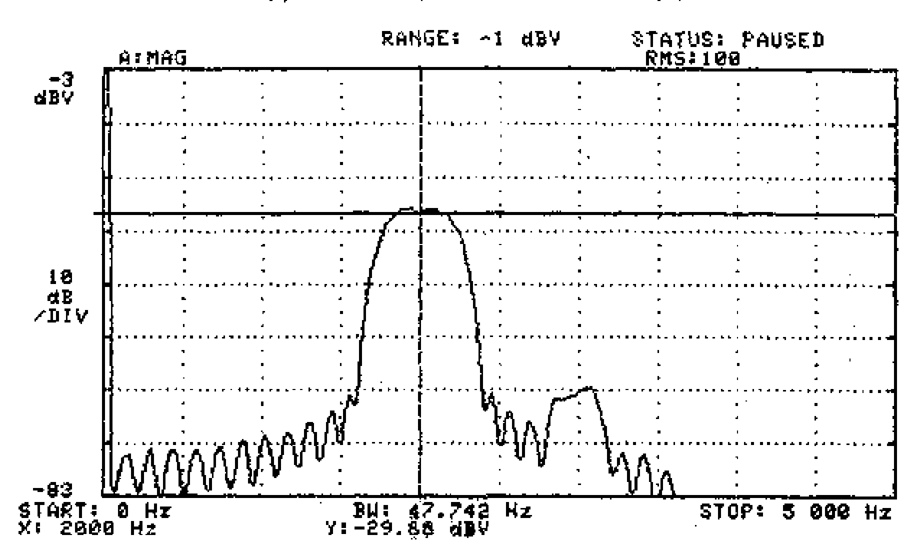
生成一个伪随机噪声序列（见例 2.16），把它作为固定系数 IIR 滤波器和自适应 FIR 滤波器的输入。自适应过程一直继续下去直到误差信号最小，反馈的误差信号是未知固定系数的 IIR 滤波器和自适应 FIR 滤波器输出的差值。

建立并运行工程 adaptIDIIR（使用 C67x 浮点工具），检验输出（adaptfir_out）收敛于（模拟）中心频率为 2 kHz 的 IIR 带通滤波器（滑动条选项最初在位置 1）。当滑动条设置在位置 2 时，检验输出（iir_out）为固定系数的 IIR 带通滤波器。

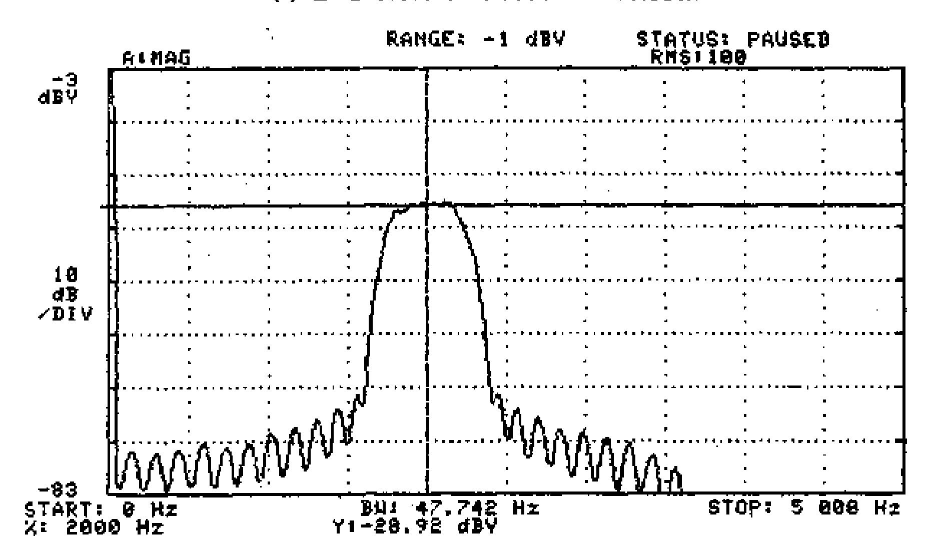
添加系数文件 lp2000.cof，替代文件 bp2000.cof，文件 lp2000.cof 表示一个截止频率为 2 kHz 的 8 阶 IIR 低通滤波器，关于该滤波器已在例 5.1 中介绍过。检验自适应 FIR 滤波器现在已调整为截止频率为 2 kHz 的 IIR 低通滤波器。



(a) 显示中心频率为 3 kHz 和 2 kHz 的滤波器



(b) 进一步收敛于中心频率为 3 kHz 的滤波器



(c) 收敛于中心频率为 3 kHz 固定系数的滤波器

图 7.14　用 HP 动态信号分析仪获得的收敛于 2 kHz 的自适应过程

```
//AdaptIDIIR.c Adaptive FIR for system ID of fixed IIR using C67x tools

#include "bp2000.cof"                   //BP @ 2kHz fixed IIR coeff
#include "noise_gen.h"                  //support file noise sequence
#define beta 1E-11                      //rate of convergence
#define WLENGTH 200                     //# of coeff for adaptive FIR
float w[WLENGTH+1];                     //buffer coeff for adaptive FIR
int dly_adapt[WLENGTH+1];               //buffer samples of adaptive FIR
int dly_fix[stages][2] = {0};           //delay samples of fixed IIR
int a[stages][3], b[stages][2];         //coefficients of fixed IIR
short out_type = 1;                     //slider adaptive FIR/fixed IIR
int fb;                                 //feedback variable for noise
shift_reg sreg;                         //shift register for noise

int prand(void)                         //pseudo-random sequence {-1,1}
{
  int prnseq;
  if(sreg.bt.b0)
      prnseq = -4000;                   //scaled negative noise level
  else
      prnseq= 4000;                     //scaled positive noise level
  fb =(sreg.bt.b0)^(sreg.bt.b1);        //XOR bits 0,1
  fb^=(sreg.bt.b11)^(sreg.bt.b13);      //with bits 11,13 ->fb
  sreg.regval<<=1;
  sreg.bt.b0=fb;                        //close feedback path
  return prnseq;                        //return noise sequence
}

interrupt void c_int11()                //ISR
{
 int i, un, input, yn;
 int iir_out=0;                         //init output of fixed IIR
 int adaptfir_out=0;                    //init output of adaptive FIR
 float E;                               //error signal

 dly_fix[0][0] = prand();               //input noise to fixed IIR
 dly_adapt[0] = dly_fix[0][0];          //same input to adaptive FIR
 input = prand();                       //noise as input to fixed IIR

 for (i = 0; i < stages; i++)           //repeat for each stage
  {
  un=input-((b[i][0]*dly_fix[i][0])>>15)-((b[i][1]*dly_fix[i][1])>>15);

  yn=((a[i][0]*un)>>15)+((a[i][1]*dly_fix[i][0])>>15)
     +((a[i][2]*dly_fix[i][1])>>15);

  dly_fix[i][1] = dly_fix[i][0];        //update delays of fixed IIR
  dly_fix[i][0] = un;                   //update delays of fixed IIR
  input = yn;                           //in next stage=out previous
  }
```

```
  iir_out = yn;                              //output of fixed IIR

  for (i = 0; i < WLENGTH; i++)
    adaptfir_out +=(w[i]*dly_adapt[i]);      //output of adaptive FIR

  E = iir_out - adaptfir_out;                //error as difference of outputs
  for (i = WLENGTH; i > 0; i--)
   {
    w[i] = w[i]+(beta*E*dly_adapt[i]);       //update weights of adaptive FIR
    dly_adapt[i] = dly_adapt[i-1];           //update samples of adaptive FIR
   }

  if (out_type == 1)                         //slider adaptive FIR/fixed IIR
    output_sample(adaptfir_out);             //output of adaptive FIR
  else if (out_type == 2)
    output_sample(iir_out);                  //output of fixed IIR
  return;                                    //return to main
}

void main()
{
  int i=0;
  for (i = 0; i < WLENGTH; i++)
   {
    w[i] = 0.0;                              //init coeff of adaptive FIR
    dly_adapt[i] = 0.0;                      //init samples of adaptive FIR
   }
  sreg.regval=0xFFFF;                        //initial seed value
  fb = 1;                                    //initial feedback value
  comm_intr();                               //init DSK, codec, McBSP
  while (1);                                 //infinite loop
}
```

图 7.15　自适应 FIR 模拟（识别）固定系数 IIR 滤波器的程序（adaptIDIIR.c）

例 7.6　自适应预测器用于消除宽带信号上叠加的窄带干扰

图 7.16 给出了实现自适应 FIR 预测器程序 adaptpredict.c，它消除宽带信号中的窄带干扰。叠加窄带干扰的宽带信号被延迟，并作为 60 个系数的自适应 FIR 滤波器的输入。

期望的宽带信号由图 7.17 所示的 MATLAB 程序 wbsignal.m 产生，它生成一个 256 点的查找表并包含在文件 wbsignal.h 中（在辅助材料中）。生成的随机序列{−1,1}被定标处理后，写入到文件 wbsignal.h 中。随机序列长为 128 位，位速率为 4 kHz，将它上抽样为位速率为 8 kHz 的 256 点序列，生成的宽带随机序列（用文件 wbsignal.h）表示所要的信号。

窄带干扰是外部信号，和产生的随机序列带宽相比（所期望的宽带信号），干扰信号的带宽相对来说较窄，因此干扰信号的样点是高度相关的，另一方面，宽带信号的样点之间相对来说是互不相关的。

```
//Adaptpredict.C Adaptive predictor to cancel interference

#include "wbsignal.h"                  //wide-band signal table look-up
#define beta 1E-14                     //rate of convergence
#define N 60                           //# of coefficients of adapt FIR
const short bufferlength = NS;         //buffer length for wideband signal
short splusn[N+1];                     //buffer wideband signal+interference
float w[N+1];                          //buffer for weights of adapt FIR
float delay[N+1];                      //buffer for input to adapt FIR

interrupt void c_int11()               //ISR
{
 static short buffercount=0;           //init buffer
 int i;
 float yn, E;                          //yn=out adapt FIR, error signal
 short wb_signal;                      //wideband desired signal
 short noise;                          //external interference

 wb_signal=wbsignal[buffercount];      //wideband signal from look-up table
 noise = input_sample();               //external input as interference
 splusn[0] = wb_signal + noise;        //wideband signal+interference
 delay[0] = splusn[3];                 //delayed input to adaptive FIR
 yn = 0;                               //init output of adaptive FIR

 for (i = 0; i < N; i++)
   yn += (w[i] * delay[i]);            //output of adaptive FIR filter
 E = splusn[0] - yn;                   //(wideband+noise)-out adapt FIR

 for (i = N-1; i >= 0; i--)
  {
   w[i] = w[i]+(beta*E*delay[i]);      //update weights of adapt FIR
   delay[i+1] = delay[i];              //update buffer delay samples
   splusn[i+1] = splusn[i];            //update buffer corrupted wideband
  }

 buffercount++;                        //incr buffer count of wideband
 if (buffercount >= bufferlength)      //if buffer count=length of buffer
   buffercount = 0;                    //reinit count
 output_sample((short)E);              //overall output
 return;
}

void main()
{
 int T = 0;
 for (T = 0; T < N; T++)               //init variables
  {
   w[T] = 0.0;                         //buffer for weights of adaptive FIR
   delay[T] = 0.0;                     //buffer for delay samples
   splusn[T] = 0;                      //buffer for wideband+interference
  }
 comm_intr();                          //init DSK, codec, McBSP
 while(1);                             //infinite loop
}
```

图7.16 消除宽带信号中窄带干扰信号的自适应预测器程序（adaptpredict.c）

```
%wbsignal.m Generates wideband random sequence. Represents one info bit

len_code = 128;                            %length of random sequence
code = 2*round(rand(1,len_code))-1;        %generates random sequence {1,-1}
sample_rate = 2;                           %up-sampling from 4 to 8kHz
NS = len_code * sample_rate;               %length of up-sampled sequence
sig = zeros(1,NS);                         %initialize random sequence
for i = 1:len_code                         %obtain up-sampled random sequence
  sig((i-1)*sample_rate + 1:i*sample_rate) = code(i);
end;
wbsignal = sig*5000;                       %scale for p-p amplitude of 500mV

fid=fopen('wbsignal.h','w');               %open file for wideband signal
fprintf(fid,'#define NS 256 //number of output sample points\n\n');
fprintf(fid,'short wbsignal[256]={');
fprintf(fid,'%d, ' ,wbsignal(1:NS-1));
fprintf(fid,'%d' ,wbsignal(NS));
fprintf(fid,'};\n\n');
fclose(fid);
return;
```

图 7.17 产生宽带随机序列的 MATLAB 程序（wbsingnal.m）

由于窄带干扰信号的特点，因此程序中根据过去的抽样 splusn 估计窄带干扰信号是可能的。叠加窄带干扰的宽带信号 splusn 先被延迟，然后输入到自适应 FIR 滤波器。延迟的时间要求足够长，使宽带信号和未延迟的抽样不相关。

自适应 FIR 滤波器的输出是相关窄带干扰的估计，因此，误差信号 E 是期望的宽带信号的估计。

建立并运行工程 adaptpredict（用 C67x 浮点工具），加入表示窄带干扰的 1 kHz 到 3 kHz 的正弦输入信号，运行程序并检验输入干扰信号逐渐减小，误差信号 E 的输出谱收敛于期望的宽带信号。

改变输入正弦干扰信号的频率，观察消除外部干扰的自适应过程，将 beta 增加到 10，可以观察到收敛速度也加快了。

期望的宽带信号可以通过输出信号 wb_signal 来观察，另外，通过 output_sample（splusn[0]），可以观察叠加干扰的宽带信号。当外部干扰信号幅度大约是宽带信号幅度的 3 倍时，能得到更好的结果。

参考文献

1. B. Widrow and S. D. Stearns, *Adaptive Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1985.
2. B. Widrow and M. E. Hoff, Jr., Adaptive switching circuits, *IRE WESCON*, 1960, pp. 96–104.
3. B. Widrow, J. R. Glover, J. M. McCool, J. Kaunitz, C. S. Williams, R. H. Hearn, J. R. Zeidler, E. Dong, Jr., and R. C. Goodlin, Adaptive noise cancelling: principles and applications, *Proceedings of the IEEE*, Vol. 63, 1975, pp. 1692–1716.
4. R. Chassaing, *Digital Signal Processing with C and the TMS320C30*, Wiley, New York, 1992.

5. D. G. Manolakis, V. K. Ingle, and S. M. Kogon, *Statistical and Adaptive Signal Processing*, McGraw-Hill, New York, 2000.
6. S. Haykin, *Adaptive Filter Theory*, Prentice Hall, Upper Saddle River, NJ, 1986.
7. J. R. Treichler, C. R. Johnson, Jr., and M. G. Larimore, *Theory and Design of Adaptive Filters*, Wiley, New York, 1987.
8. S. M. Kuo and D. R. Morgan, *Active Noise Control Systems*, Wiley, New York, 1996.
9. K. Astrom and B. Wittenmark, *Adaptive Control*, Addison-Wesley, Reading, MA, 1995.
10. J. Tang, R. Chassaing, and W. J. Gomes III, Real-time adaptive PID controller using the TMS320C31 DSK, *Proceedings of the 2000 Texas Instruments DSPS Fest Conference*, 2000.
11. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, Wiley, New York, 1999.
12. R. Chassaing et al., Student projects on applications in digital signal processing with C and the TMS320C30, *Proceedings of the 2nd Annual TMS320 Educators Conference*, Texas Instruments, Dallas, TX, 1992.
13. C. S. Linquist, *Adaptive and Digital Signal Processing*, Steward and Sons, 1989.
14. S. D. Stearns and D. R. Hush, *Digital Signal Analysis*, Prentice Hall, Upper Saddle River, NJ, 1990.
15. J. R. Zeidler, Performance analysis of LMS adaptive prediction filters, *Proceedings of the IEEE*, Vol. 78, 1990, pp. 1781–1806.
16. S. T. Alexander, *Adaptive Signal Processing: Theory and Applications*, Springer-Verlag, New York, 1986.
17. C. F. Cowan and P. F. Grant, eds., *Adaptive Filters*, Prentice Hall, Upper Saddle River, NJ, 1985.
18. M. L. Honig and D. G. Messerschmitt, *Adaptive Filters: Structures, Algorithms and Applications*, Kluwer Academic, Norwell, MA, 1984.
19. V. Solo and X. Kong, *Adaptive Signal Processing Algorithms: Stability and Performance*, Prentice Hall, Upper Saddle River, NJ, 1995.
20. S. Kuo, G. Ranganathan, P. Gupta, and C. Chen, Design and implementation of adaptive filters, *IEEE 1988 International Conference on Circuits and Systems*, June 1988.
21. M. G. Bellanger, *Adaptive Digital Filters and Signal Analysis*, Marcel Dekker, New York, 1987.
22. R. Chassaing and B. Bitler, Adaptive filtering with C and the TMS320C30 digital signal processor, *Proceedings of the 1992 ASEE Annual Conference*, June 1992.
23. R. Chassaing, D. W. Horning, and P. Martin, Adaptive filtering with the TMS320C25, *Proceedings of the 1989 ASEE Annual Conference*, June 1989.

第8章 程序优化方法

本章内容主要包括：（1）提高程序效率的优化方法；（2）内部C函数；（3）并行指令；（4）字长数据的访问；（5）软件流水线方法。

本章将讲述几种程序优化方法，这些方法将极大地降低程序执行时间，包括并行执行指令的使用、按字访问数据、内部函数和软件流水线方法等。

8.1 引言

我们从工作站级开始，例如在PC上使用C程序。汇编语言程序只适合于特定的处理器硬件，但C语言程序却可以很容易地从一个平台移植到另一个平台；另一方面，经过优化的汇编语言程序要比C语言程序执行速度快，且占用的内存更少。

程序优化之前，首先要确保程序可以实现相应的功能并能得到正确的结果。因为程序经优化后，程序代码的顺序可能要被重新组织和排序，这样，优化过程会使程序变得很难阅读和理解。在程序优化时，必须意识到：如果C程序算法的功能是正确的且运算速度令人满意，就没有必要再对程序进行优化。

在测试完C程序的功能后，把它传输到C6x平台上。先用浮点方式进行模拟仿真，如果需要的话，再把它转换成定点实现方式。如果程序的性能不够好的话，再利用不同的编译选项实现软件流水线（后面将讨论），减少冗余循环等。如果程序的性能仍达不到要求，可以不使用循环语句，避免分支语句中的开销所占用时间，这种措施通常可以提高运算速度，但会增加程序的长度。为了提高运行速度，也可使用按字进行存取的优化方法，即存取32位（int整型数据）的字而不是16位（short短整型数据）的半字，然后再分别对高低16位数据进行处理。

如果性能仍不能令人满意，可以用线性汇编重新编写对时间要求较严的程序部分，进而用汇编优化程序进行优化，也可以使用函数剖析器确定需要进一步优化的特定函数。

最后一个讨论的优化过程是软件流水线方法，通过该技术得到手工编制的汇编指令[1,2]。为了获得高效最优的程序，遵循软件流水线相关的步骤是非常重要的。

8.2 优化步骤

如果经过任何特定的步骤，程序的性能和结果是令人满意的，那么优化过程也就结束了。一般来说，需要经过以下几步：

1. 编写C语言程序，创建未经优化的相关工程。
2. 在不同优化层次上进行优化。如果合适的话，使用内部函数。
3. 使用剖析器确定需要进一步优化的函数，然后将它们修改成线性汇编函数。
4. 对汇编程序进行优化。

8.2.1 编译器选项

当优化程序启动后，它将按下面的步骤执行：C 程序首先通过语法解析程序，执行预处理功能，并产生一个中间文件（.if）输入给优化程序，此后优化程序就会生成一个.opt 文件，并将它输入给程序代码产生器进行进一步优化，最后生成汇编语言文件。

编译器选项的不同，表示启用不同的优化方法。编译选项包括：

1. –o0 优化寄存器的使用。
2. –o1 除执行前一选项–o0 的优化功能外，还进行局部优化。
3. –o2 除执行前面两个选项–o0 和–o1 的优化功能外，还进行全局优化。
4. –o3 除执行前面三个选项–o0，–o1 和–o2 优化功能外，还对整个文件进行优化。

选项–o2，–o3 还试图进行软件优化。

8.2.2 内部 C 函数

有许多现成的 C 语言内部函数可供调用，这些函数可提高程序的效率（参见例 3.1）：

1. int_mpy( )与汇编指令中的 MPY 相同，将一个数的 16 位最低有效位乘以另一个数的 16 位最低有效位。
2. int_mpyh( )与汇编指令中的 MPYH 相同，将一个数的 16 位最高有效位乘以另一个数的 16 位最高有效位。
3. int_mpylh( )与汇编指令中的 MPYLH 相同，将一个数的 16 位最低有效位乘以另一个数的 16 位最高有效位。
4. int_mpyhl( )与汇编指令中的 MPYHL 相同，将一个数的 16 位最高有效位乘以另一个数的 16 位最低有效位。
5. void_nassert(int)不产生程序代码，它通知编译器用表达式说明的声明函数为真，并把相关信息通知给编译器。这些信息包括：指针、数据的排列方式以及有效的优化方案，如字长优化的排列方式。
6. uint_lo(double)与 uint_hi(double)分别获得双精度字的高 32 位与低 32 位（只在 C67x 或 C64x 才有）。

8.3 代码的优化过程

1. 使用并行指令以便在同一个时钟周期操作多个功能单元。
2. 删除空操作指令（NOP）或延迟时隙，在 NOP 处放置程序代码。
3. 解开循环语句，避免分支语句的开销。
4. 使用按字访问 32 位字（int），而不是 16 位半字（short）。
5. 使用软件流水线，在 8.5 节将对此进行介绍。

8.4 使用代码优化方法的程序举例

下面将举例说明不同的优化方法是如何提高代码效率的（利用软件流水线进行优化将在 8.5 节

讨论），仍然使用点积作为例子说明不同优化方案的作用。两个数组的点积运算对于许多 DSP 算法来说是非常有益的，如滤波与相关运算。假定在下面的例子中，每个数组包含 200 个数。第 3 章给出了一些使用 C 和汇编的混合编程的例子，提供了一些必要的背景。

例 8.1　按字访问数据，定点实现乘积之和的 C 程序

图 8.1 为 C 程序 twosum.c，它可按 32 位字格式访问数据，实现两个数组的乘积之和。例中每个数组含有 200 个数。在循环中分别计算偶数项和奇数项各自乘积的和，在循环外计算奇数项与偶数项的总和。

由于是浮点型，图 8.1 中的函数及变量 sum，suml 和 sumh 定义为 float，而不是 int：

```
float  dotp (float a[ ], float b [ ])
{
       float suml, sumh, sum;
       int i;
       .
       .
       .
}
```

```
//twosum.c Sum of Products with separate accumulation of even/odd terms
//with word-wide data for fixed-point implementation

int     dotp (short a[ ], short b [ ])
 {
        int suml, sumh, sum, i;
        suml = 0;
        sumh = 0;
        sum = 0;

        for (i = 0; i < 200; i +=2)
         {
          suml += a[i] * b[i];                  //sum of products of even terms
          sumh += a[i + 1] * b[i + 1];          //sum of products of odd terms
         }
        sum = suml + sumh;                      //final sum of odd and even terms
        return (sum);
}
```

图 8.1　按字访问数据，对乘积偶数项和奇数项单独求和，实现乘积之和的 C 程序（twosum.c）

例 8.2　利用 C 内部函数实现乘积不同部分的和

图 8.2 为 C 程序 dotpintrinsic.c，它说明了利用 C 内部函数_mpy 与_mpyh 来计算各自乘积的和，这两个函数在汇编语言函数中对应等价于 MPY 和 MPYH。尽管如此，奇数项与偶数项乘积之和仍在循环内计算的，最后总和的计算被放在循环之外，作为函数的返回值。

```
//dotpintrinsic.c  Sum of products with C intrinsic functions using C

for (i = 0; i < 100; i++)
      {
            suml = suml + _mpy(a[i], b[i]);
            sumh = sumh + _mpyh(a[i], b[i]);
      }
return (suml + sumh);
```

图 8.2 利用 C 内部函数计算各自乘积的和（dotpintrinsic.c）

例 8.3 利用线性汇编程序及按字访问数据，定点实现乘积的和

图 8.3 为线性汇编程序 twosumlasmfix.sa，它利用线性汇编定点实现乘积的两个不同部分的和。程序中没有必要指定功能单元和 NOP 指令，另外，符号名可用来表示寄存器。LDW 指令可用来传送一个 32 位字长的数据（使用 LDW 时，数据在存储器中一定要按字对齐排列）。低 16 位和高 16 位乘积单独计算，两个 ADD 指令分别累加奇数项与偶数项积的和。

```
;twosumlasmfix.sa  Sum of Products. Separate accum of even/odd terms
;With word-wide data for fixed-point implementation using linear ASM

loop:        LDW      *aptr++, ai           ;32-bit word ai
             LDW      *bptr++, bi           ;32-bit word bi
             MPY      ai, bi, prodl         ;lower 16-bit product
             MPYH     ai, bi, prodh         ;higher 16-bit product
             ADD      prodl, suml, suml     ;accum even terms
             ADD      prodh, sumh, sumh     ;accum odd terms
             SUB      count, 1, count       ;decrement count
   [count]   B        loop                  ;branch to loop
```

图 8.3 利用线性汇编，定点实现乘积不同部分的和（twosumlasmfix.sa）

例 8.4 利用线性汇编及双精度数传送指令，浮点方式实现乘积的和

图 8.4 为线性汇编程序 twosumlasmfloat.sa，它利用线性汇编及双精度数传送指令，浮点实现乘积的和。双精度数传送指令 LDDW 将一个 64 位的数据传送到一寄存器对中。每个单精度乘积指令 MPYSP 执行一个 32×32 的乘法，低 32 位与高 32 位的乘积之和得到 32 位奇偶项的总和。

```
;twosumlasmfloat.sa  Sum of products. Separate accum of even/odd terms
;Using double-word load LDDW for floating-point implementation

loop:        LDDW     *aptr++, ai1:ai0      ;64-bit word ai0 and ai1
             LDDW     *bptr++, bi1:bi0      ;64-bit word bi0 and bi1
             MPYSP    ai0, bi0, prodl       ;lower 32-bit product
             MPYSP    ai1, bi1, prodh       ;hiagher 32-bit product
             ADDSP    prodl, suml, suml     ;accum 32-bit even terms
             ADDSP    prodh, sumh, sumh     ;accum 32-bit odd terms
             SUB      count, 1, count       ;decrement count
   [count]   B        loop                  ;branch to loop
```

图 8.4 利用线性汇编及 LDDW 指令，浮点实现乘积不同部分的和（twosumlasmfloat.sa）

例 8.5　使用汇编及无并行指令，定点实现点积

图 8.5 为汇编程序 dotpnp.asm，该程序没有使用并行指令，并采用定点方式实现点积。所有的 C6x 器件都可执行定点指令，但浮点实现需要像 C6711 DSK 这样的 C67x 平台。

```
;dotpnp.asm   ASM Code with no-parallel instructions for fixed-point

              MVK        .S1    200, A1      ;count into A1
              ZERO       .L1    A7           ;init A7 for accum

LOOP          LDH        .D1    *A4++,A2     ;A2=16-bit data pointed by A4
              LDH        .D1    *A8++,A3     ;A3=16-bit data pointed by A8
              NOP               4            ;4 delay slots for LDH
              MPY        .M1    A2,A3,A6     ;product in A6
              NOP                            ;1 delay slot for MPY
              ADD        .L1    A6,A7,A7     ;accum in A7
              SUB        .S1    A1,1,A1      ;decrement count
      [A1]    B          .S2    LOOP         ;branch to LOOP
              NOP               5            ;5 delay slots for B
```

图 8.5　无并行指令，定点实现点积的汇编程序（dotpnp.asm）

一个循环可以迭代 200 次。对于定点实现，每个指针寄存器 A4 和 A8 加 1，指向缓冲区的下一个半字（16 位），但是对于浮点实现，一个指针寄存器加 1，指向的是下一个 32 位的字。数据传送、相乘和分支指令必须分别使用功能单元.D，.M 和.S；加法与减法指令可以使用任何单元（除.M 外）。在循环内的指令每次迭代需要 16 个时钟周期，这样就需要 $16 \times 200 = 3200$ 个时钟周期。表 8.4（见本章结尾）给出了定点和浮点实现的几种优化方案的比较情况。

例 8.6　使用汇编和并行指令，定点实现点积

图 8.6 是汇编程序 dotpp.asm，它使用并行指令，定点实现点积。它使用程序代码替代 NOP 指令，从而减少 NOP 指令的数量。

由于两个操作数来自不同的寄存器组或不同的路径，MPY 指令使用一个交叉路径（.Mlx）。SUB 和 B 指令被移到程序前面，填补 LDH 指令需要的延迟时隙；分支指令在 ADD 指令之后执行。由于使用并行指令，在循环内的指令现在只需要 8 个时钟周期，一共需要 $8 \times 200 = 1600$ 时钟周期。

```
;dotpp.asm  ASM Code with parallel instructions for fixed-point

             MVK       .S1    200, A1      ;count into A1
     ||      ZERO      .L1    A7           ;init A7 for accum

LOOP         LDH       .D1    *A4++,A2     ;A2=16-bit data pointed by A4
     ||      LDH       .D2    *B4++,B2     ;B2=16-bit data pointed by B4
             SUB       .S1    A1,1,A1      ;decrement count
     [A1]    B         .S1    LOOP         ;branch to LOOP (after ADD)
             NOP              2            ;delay slots for LDH and B
             MPY       .M1x   A2,B2,A6     ;product in A6
             NOP                           ;1 delay slot for MPY
             ADD       .L1    A6,A7,A7     ;accum in A7,then branch
;branch occurs here
```

图 8.6　使用并行指令，定点实现点积的汇编程序（dotpp.asm）

例 8.7 使用汇编程序及按字访问数据（32 位），定点实现乘积的两个不同部分的和

图 8.7 是汇编程序 twosumfix.asm，它按字访问数据，采用定点方式，计算乘积的两个不同部分的和。由于每次迭代可算出两个乘积的和，循环次数初始值设为 100（而不是 200）。指令 LDW 传送一个字或一个 32 位的数据。乘法指令 MPY 求低 16 × 16 位数据的积，MPYH 求高 16 × 16 位数据的积。两个 ADD 指令分别计算奇数项和偶数项积的和。注意在循环外面需要附加一条 ADD 指令来累加寄存器 A7 和 B7 的和，循环内的指令需要 8 个时钟周期，因此现在 100 次迭代（不是 200）共需 8 × 100 = 800 个时钟周期。

```
;twosumfix.asm ASM code for two sums of products with word-wide data
;for fixed-point implementation

            MVK     .S1    100, A1       ;count/2 into A1
   ||       ZERO    .L1    A7            ;init A7 for accum of even terms
   ||       ZERO    .L2    B7            ;init B7 for accum of odd terms

LOOP        LDW     .D1    *A4++,A2      ;A2=32-bit data pointed by A4
   ||       LDW     .D2    *B4++,B2      ;A3=32-bit data pointed by B4
            SUB     .S1    A1,1,A1       ;decrement count
      [A1]  B       .S1    LOOP          ;branch to LOOP (after ADD)
            NOP            2             ;delay slots for both LDW and B
            MPY     .M1x   A2,B2,A6      ;lower 16-bit product in A6
   ||       MPYH    .M2x   A2,B2,B6      ;upper 16-bit product in B6
            NOP                          ;1 delay slot for MPY/MPYH
            ADD     .L1    A6,A7,A7      ;accum even terms in A7
   ||       ADD     .L2    B6,B7,B7      ;accum odd terms in B7
;branch occurs here
```

图 8.7 定点实现 32 位数据乘积两部分和的汇编程序（twosumfix.asm）

例 8.8 不采用并行指令，利用汇编程序和浮点方式实现点积

图 8.8 是汇编程序 dotpnpfloat.asm，该程序没有并行指令，采用浮点方式实现点积运算。循环迭代 200 次。单精度浮点指令 MPYSP 执行 32 × 32 位的乘法，每个 MPYSP 和 ADDSP 指令需要三个延迟时隙，循环中的指令每次迭代总共需要 18 个时钟周期（不包括与 ADDSP 相关的三个 NOP 指令），这样总共需要 18 × 200 = 3600 个时钟周期。（参见表 8.4，它概括了定点与浮点实现的几种优化方案性能。）

```
;dotpnpfloat.asm ASM with no parallel instructions for floating-point

            MVK     .S1    200, A1       ;count into A1
            ZERO    .L1    A7            ;init A7 for accum

LOOP        LDW     .D1    *A4++,A2      ;A2=32-bit data pointed by A4
            LDW     .D1    *A8++,A3      ;A3=32-bit data pointed by A8
            NOP            4             ;4 delay slots for LDW
            MPYSP   .M1    A2,A3,A6      ;product in A6
            NOP            3             ;3 delay slots for MPYSP
            ADDSP   .L1    A6,A7,A7      ;accum in A7
            SUB     .S1    A1,1,A1       ;decrement count
      [A1]  B       .S2    LOOP          ;branch to LOOP
            NOP            5             ;5 delay slots for B
```

图 8.8 没有并行指令，采用浮点方式实现点积的汇编程序（dotpnpfloat.asm）

例 8.9 使用汇编程序及并行指令，浮点实现点积

图 8.9 是汇编程序 dotppfloat.asm，它使用并行指令，采用浮点方式实现点积。循环迭代 200 次。通过将 SUB 和 B 指令移到程序前面代替一些 NOP 指令，使得循环中指令的数目降到 10 条。注意：在循环外面需要三个附加的 NOP 指令以得到 ADDSP 的结果。循环中的指令每次迭代需要 10 个时钟周期，因此总共需要 $10\times200=2000$ 次时钟周期。

```
;dotppfloat.asm ASM Code with parallel instructions for floating-point

         MVK      .S1    200, A1        ;count into A1
  ||     ZERO     .L1    A7             ;init A7 for accum

LOOP     LDW      .D1    *A4++,A2       ;A2=32-bit data pointed by A4
  ||     LDW      .D2    *B4++,B2       ;B2=32-bit data pointed by B4
         SUB      .S1    A1,1,A1        ;decrement count
         NOP             2              ;delay slots for both LDW and B
   [A1]  B        .S2    LOOP           ;branch to LOOP (after ADDSP)
         MPYSP    .M1x   A2,B2,A6       ;product in A6
         NOP             3              ;3 delay slots for MPYSP
         ADDSP    .L1    A6,A7,A7       ;accum in A7,then branch
;branch occurs here
```

图 8.9 使用并行指令，采用浮点方式实现点积的汇编程序（dotppfloat.asm）

例 8.10 使用汇编程序及双字（64 位）数据访问方法，浮点实现乘积两部分的和

图 8.10 是汇编程序 twosumfloat.asm，它使用双字数据访问方法，浮点实现的乘积两部分之和。由于一次迭代可以算出两个乘积的和，循环计数器初始值设为 100。指令 LDDW 将一个 64 位双字数据传送到一寄存器对中；乘法指令 MPYSP 执行 32×32 位乘法；两个 ADDSP 指令分别计算奇数项和偶数项积的和。在循环外面需要附加一个 ADDSP 指令来累加寄存器 A7 与 B7 的和。一个循环中的指令共需 10 个时钟周期，现在迭代 100 次（不是 200 次），共需 $10\times100=1000$ 个时钟周期。

```
;twosumfloat.asm ASM Code for two sums of products for floating-point

         MVK      .S1    100, A1          ;count/2 into A1
  ||     ZERO     .L1    A7               ;init A7 for accum of even terms
  ||     ZERO     .L2    B7               ;init B7 for accum of odd terms

LOOP     LDDW     .D1    *A4++,A3:A2      ;64-bit into register pair A2,A3
  ||     LDDW     .D2    *B4++,B3:B2      ;64-bit into register pair B2,B3
         SUB      .S1    A1,1,A1          ;decrement count
         NOP             2                ;delay slots for LDW
   [A1]  B        .S2    LOOP             ;branch to LOOP
         MPYSP    .M1x   A2,B2,A6         ;lower 32-bit product in A6
  ||     MPYSP    .M2x   A3,B3,B6         ;upper 32-bit product in B6
         NOP             3                ;3 delay slot for MPYSP
         ADDSP    .L1    A6,A7,A7         ;accum even terms in A7
  ||     ADDSP    .L2    B6,B7,B7         ;accum odd terms in B7
;branch occurs here
         NOP             3                ;delay slots for last ADDSP
         ADDSP    .L1x   A7,B7,A4         ;final sum of even and odd terms
         NOP             3                ;delay slots for ADDSP
```

图 8.10 采用浮点方式实现乘积两部分的和的汇编程序（twosumfloat.asm）

8.5 程序优化的软件流水线方法

软件流水线方法是一种高效的汇编语言编程方法，它可使在一个时钟周期利用所有的功能单元。–o2 和–o3 优化级别可使程序代码生成（或试图生成）软件流水线程序代码。

软件流水线方法有下面三个相关的阶段：

1. 开始部分（准备部分）。该阶段包含建立循环内核（周期）所需的指令。
2. 循环内核（周期）。在循环中，所有指令都是并行执行的。由于所有指令在循环内核阶段都是并行的，所以整个循环内核在一个时钟周期内执行。
3. 结尾部分（收尾部分）。该阶段包含完成所有迭代所必需的指令。

8.5.1 手工编制软件流水线程序的过程

1. 绘制关联图。
2. 建立进程时序表。
3. 由进程时序表获得程序代码。

8.5.2 关联图

图 8.11 给出了一幅关联图。绘制一幅关联图的步骤如下：

1. 画出节点和路径。
2. 写出完成一条指令所需的时钟周期数。
3. 给每个节点分配功能单元。
4. 分离数据路径，以便使用最大数目的功能单元。

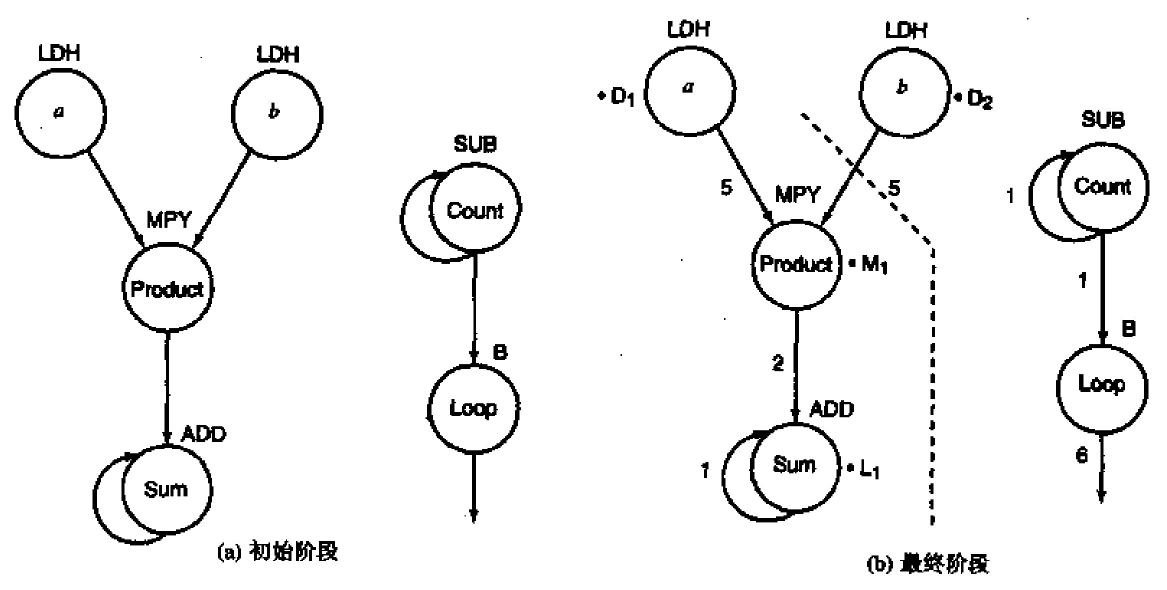


图 8.11 点积的关联图

一个节点有一个或多个数据路径进入或离开，临近节点的数字表示完成相关指令所需的时钟周期数。一个父节点包含一条写入变量的指令，而一个子节点包含一条读取变量指令，该变量是由父节点写入的。

由于两条 LDH 指令的结果被 MPY 指令利用，所以 LDH 指令被认为是 MPY 指令的父节点；类

似地，MPY 为 ADD 指令的父节点。ADD 指令反馈回来作为下次迭代的输入， SUB 指令也类似。

图 8.12 给出了另一幅关联图，它用定点实现乘积的两部分之和。在图 8.12 中，开始部分路径长度是关联图中路径最长的一部分。因为最长的路径长度为 8，开始部分在第 8 时钟进入循环内核（周期）之前，因此开始部分的长度为 7。

类似地，用 LDW，MPYSP 和 ADDSP 指令分别代替图 8.12 中的 LDH，MPY 和 ADD 指令，可获得浮点实现的关联图。注意，单精度指令 MPYSP 和 ADDSP 都需要 4 个时钟周期来执行完（每条指令需要三个延迟时隙）。

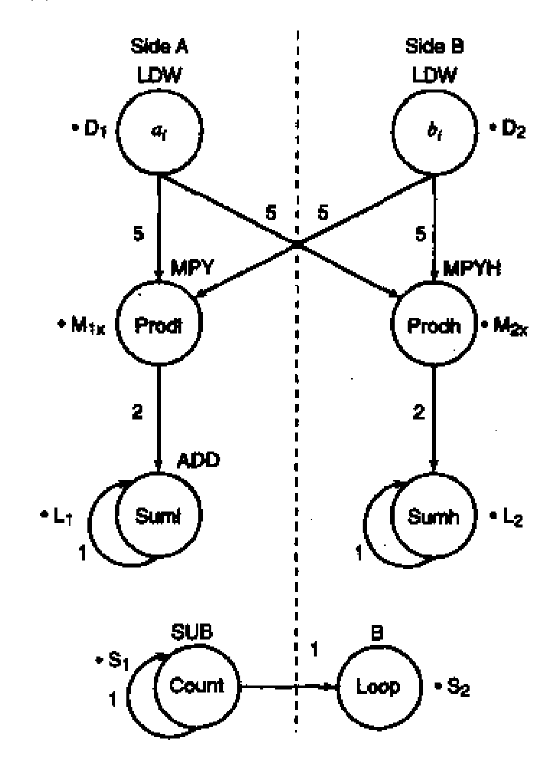


图 8.12 每次迭代乘积的两个部分之和的关联图

8.5.3 进程时序表

表 8.1 是依据关联图所做出的进程时序表。

1. LDW 从第 1 个时钟周期开始执行。
2. 由于有 4 个延迟时隙，MPY 和 MPYH 指令必须在 LDW 之后 5 个时钟周期开始执行，因此 MPY 和 MPYH 在第 6 个时钟开始执行。
3. 由于有 1 个延迟时隙，ADD 指令必须在 MPY/MPYH 指令之后 2 个时钟周期开始执行，因此 ADD 指令在第 8 个时钟周期开始执行。
4. B 指令有 5 个延迟时隙，它在第 3 个时钟周期开始执行，分支转移发生在第 9 个时钟周期开始，即在 ADD 指令后。
5. SUB 指令必须分支指令前一个时钟周期开始执行，因为在分支转移产生之前，循环计数器减 1，所以 SUB 指令在第 2 个时钟周期开始执行。

从表 8.1 可以看出，两条 LDW 指令是并行执行的，分别在第 1, 9, 17, …时钟周期执行，SUB 指令在第 2, 10, 18, …时钟周期执行，紧接着是分支指令（B）在第 3, 11, 19, …时钟周期执行，两个并行指令 MPY 和 MPYH 在第 6, 14, 22, …时钟周期执行，ADD 指令在第 8, 16, 24, …时钟周期执行。

表8.1　利用软件流水线方法前，定点实现点积的进程时序表

单元	周期 1,9,…	2,10,…	3,11,…	4,12,…	5,13,…	6,14,…	7,15,…	8,16,…
.D1	LDW							
.D2	LDW							
.M1						MPY		
.M2						MPYH		
.L1								ADD
.L2								ADD
.S1		SUB						
.S2			B					

为了说明三个不同阶段：开始部分（第1到第7时钟周期）、内核部分（第8时钟周期）和结尾部分（第9, 10, …时钟周期，不一一列出），将表8.1扩展成表8.2所示的形式。开始部分和循环内的指令依次重复执行，结尾部分的指令（第9, 10, …时钟周期）完成程序的最终功能。

从表8.2可获得高效的程序代码。注意，在前面的迭代完成之前，有可能开始处理新的迭代。使用软件流水线方法可确定下一个新的循环迭代开始时间。

表8.2　利用软件流水线方法后，定点实现点积的进程时序表

单元	周期 开始阶段 1	2	3	4	5	6	7	循环内核 8
.D1	LDW	LDW	LDW	LDW	LDW	LDW	LDW	LDW
.D2	LDW	LDW	LDW	LDW	LDW	LDW	LDW	LDW
.M1						MPY	MPY	MPY
.M2						MPYH	MPYH	MPYH
.L1								ADD
.L2								ADD
.S1		SUB	SUB	SUB	SUB	SUB	SUB	SUB
.S2			B	B	B	B	B	B

循环内核（周期）

在循环内核内第8个时钟周期，每个功能单元只使用一次。最小迭代间隔时间是指开始下一个迭代之前所需等待的最小时钟周期数。由于该时间间隔是1个时钟周期，因此每个时钟周期启动一次新的迭代。

在循环内核内第8个时钟周期，循环的多次迭代并行执行。在第8个时钟周期的相同时间里处理不同的迭代。例如，ADD数据相加指令是第1次迭代的指令，而MPY和MPYH数据相乘指令是第3次迭代的指令，LDW数据传送指令是第8次迭代的指令，SUB指令在第7次迭代里使计数器减1，B分支指令是第6次迭代里的指令。注意，要相乘的数据在数据相乘周期前5个时钟周期要送到寄存器中。在第1个乘法相乘前，第5个将要相乘的数据就已经传送到寄存器了，该软件流水线的深度是8次迭代。

例8.11　利用软件流水线方法定点实现点积

该例使用软件流水线方法定点实现点积的运算。从表8.2可以很容易获得如图8.13所示的汇

编程序 dotpipedfix.asm。循环次数是 100 次，因为每次迭代分别有两个乘加运算。下面指令分别是在如下不同的时钟周期开始执行：

```
;dotpipedfix.asm  ASM code for dot product with software pipelining
;For fixed-point implementation
;cycle 1
              MVK       .S1     100,A1        ;loop count
         ||   ZERO      .L1     A7            ;init accum A7
         ||   ZERO      .L2     B7            ;init accum B7
         ||   LDW       .D1     *A4++,A2      ;32-bit data in A2
         ||   LDW       .D2     *B4++,B2      ;32-bit data in B2
;cycle 2
         ||   LDW       .D1     *A4++,A2      ;32-bit data in A2
         ||   LDW       .D2     *B4++,B2      ;32-bit data in B2
         || [A1] SUB    .S1     A1,1,A1       ;decrement count
;cycle 3
         ||   LDW       .D1     *A4++,A2      ;32-bit data in A2
         ||   LDW       .D2     *B4++,B2      ;32-bit data in B2
         || [A1] SUB    .S1     A1,1,A1       ;decrement count
         || [A1] B      .S2     LOOP          ;branch to LOOP
;cycle 4
         ||   LDW       .D1     *A4++,A2      ;32-bit data in A2
         ||   LDW       .D2     *B4++,B2      ;32-bit data in B2
         || [A1] SUB    .S1     A1,1,A1       ;decrement count
         || [A1] B      .S2     LOOP          ;branch to LOOP
;cycle 5
         ||   LDW       .D1     *A4++,A2      ;32-bit data in A2
         ||   LDW       .D2     *B4++,B2      ;32-bit data in B2
         || [A1] SUB    .S1     A1,1,A1       ;decrement count
         || [A1] B      .S2     LOOP          ;branch to LOOP
;cycle 6
         ||   LDW       .D1     *A4++,A2      ;32-bit data in A2
         ||   LDW       .D2     *B4++,B2      ;32-bit data in B2
         || [A1] SUB    .S1     A1,1,A1       ;decrement count
         || [A1] B      .S2     LOOP          ;branch to LOOP
         ||   MPY       .M1x    A2,B2,A6      ;lower 16-bit product into A6
         ||   MPYH      .M2x    B2,A2,B6      ;upper 16-bit product into B6
;cycle 7
         ||   LDW       .D1     *A4++,A2      ;32-bit data in A2
         ||   LDW       .D2     *B4++,B2      ;32-bit data in B2
         || [A1] SUB    .S1     A1,1,A1       ;decrement count
         || [A1] B      .S2     LOOP          ;branch to LOOP
         ||   MPY       .M1x    A2,B2,A6      ;lower 16-bit product into A6
         ||   MPYH      .M2x    B2,A2,B6      ;upper 16-bit product into B6
;cycles 8-107 (loop cycle)
         ||   LDW       .D1     *A4++,A2      ;32-bit data in A2
         ||   LDW       .D2     *B4++,B2      ;32-bit data in B2
         || [A1] SUB    .S1     A1,1,A1       ;decrement count
         || [A1] B      .S2     LOOP          ;branch to LOOP
         ||   MPY       .M1x    A2,B2,A6      ;lower 16-bit product into A6
         ||   MPYH      .M2x    B2,A2,B6      ;upper 16-bit product into B6
         ||   ADD       .L1     A6,A7,A7      ;accum in A7
         ||   ADD       .L2     B6,B7,B7      ;accum in B7
;branch occurs here
;cycle 108 (epilog)
              ADD       .L1x    A7,B7,A4      ;final accum of odd/even
```

图 8.13　利用软件流水线方法，定点实现点积的汇编程序（dotpipedfix.asm）

时钟周期 1：LDW，LDW（也包括计数器、累加器 A7 和 B7 的初始化）
时钟周期 2：LDW，LDW，SUB
时钟周期 3-5：LDW，LDW，SUB，B
时钟周期 6-7：LDW，LDW，MPY，MPYH，SUB，B
时钟周期 8-107：LDW，LDW，MPY，MPYH，ADD，ADD，SUB，B
时钟周期 108：LDW，LDW，MPY，MPYH，ADD，ADD，SUB，B

开始部分执行从第 1 到 7 个时钟周期；循环内核部分从第 8 个时钟周期开始执行，这里所有的指令都是并行执行的；结尾部分在第 108 个时钟周期。注意：SUB 指令是有条件的，保证 A1 减到 0 后不再继续递减。

例 8.12 利用软件流水线方法浮点实现点积

本例采用软件流水线方法浮点实现点积运算，表 8.3 中给出了表 8.2 的浮点实现形式。相对于表 8.2，LDW 变为 LDDW，MPY/MPYH 变为 MPYSP，ADD 变为 ADDSP，MPYSP 和 ADDSP 都各有 3 个延迟时隙，因此，循环内核从第 10 个时钟周期开始执行（而不是从第 8 个时钟周期开始执行）。SUB 和 B 指令分别从第 4 个和第 5 个时钟周期开始执行，而不是从第 2 个和第 3 个时钟周期。ADDSP 从第 10 个时钟周期开始执行，而不是从第 8 个时钟周期，因此采用浮点实现方式，流水线深度为 10。

表 8.3 利用软件流水线方法后，浮点实现点积的进程时序表

	周期									
	开始阶段									循环内核
单元	1	2	3	4	5	6	7	8	9	10
.D1	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW
.D2	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW
.M1						MPYSP	MPYSP	MPYSP	MPYSP	MPYSP
.M2						MPYSP	MPYSP	MPYSP	MPYSP	MPYSP
.L1										MPYSP
.L2										MPYSP
.S1				SUB	SUB	SUB	SUB	SUB	SUB	SUB
.S2					B	B	B	B	B	B

图 8.14 给出了利用软件流水线技术浮点实现点积的汇编程序 dotpipedfloat.asm，由于 ADDSP 指令有 3 个延迟时隙，因此累加的各项序号之间相差 4。在每个循环周期，ADDSP 指令相关的累加内容如下：

循环周期	累加项 (单个 ADDSP)	
1	0	
2	0	
3	0	
4	0	
5	p0	;第 1 个乘积
6	p1	;第 2 个乘积

```
7         p3
8         p4
9         p0 + p4                        ;第1个和第5个乘积的和
10        p1 + p5                        ;第2个和第6个乘积的和
11        p2 + p6
12        p3 + p7
13        p0 + p4 + p8                   ;第1个、第5个和第9个乘积的和
14        p1 + p5 + p9
15        p2 + p6 + p10
16        p3 + p7 + p11
17        p0 + p4 + p8 + p12
.                .
.                .
.                .
99        p2 + p6 + p10 + … + p94
100       p3 + p7 + p11 + … + p95
```

对应每个循环周期，给出了各周期累加的过程，实际的循环周期偏移了9个时钟周期（开始部分所占的时钟周期）。注意的是：因为第一个ADDSP指令从第1个循环周期开始且有3个延迟时隙，因此第一个乘积p0在第5个循环周期才能得到，它和低32位项有关，而第2个ADDSP（未显示出来）累加高32位部分的乘积之和。

A6包含低32位的积而B6包含高32位的积。高低32位乘积的和分别在A7和B7中累积。

结尾部分的实际时钟周期（不是指循环周期）包括下列指令，如图8.14所示。

```
;dotpipedfloat.asm ASM code for dot product with software pipelining
;For floating-point implementation
;cycle 1
              MVK      .S1     100,A1        ;loop count
        ||    ZERO     .L1     A7            ;init accum A7
        ||    ZERO     .L2     B7            ;init accum B7
        ||    LDDW     .D1     *A4++,A3:A2   ;64-bit data in A2 and A3
        ||    LDDW     .D2     *B4++,B3:B2   ;64-bit data in B2 and B3
;cycle 2
        ||    LDDW     .D1     *A4++,A3:A2   ;64-bit data in A2 and A3
        ||    LDDW     .D2     *B4++,B3:B2   ;64-bit data in B2 and B3
;cycle 3
        ||    LDDW     .D1     *A4++,A3:A2   ;64-bit data in A2 and A3
        ||    LDDW     .D2     *B4++,B3:B2   ;64-bit data in B2 and B3
;cycle 4
        ||    LDDW     .D1     *A4++,A3:A2   ;64-bit data in A2 and A3
        ||    LDDW     .D2     *B4++,B3:B2   ;64-bit data in B2 and B3
        || [A1] SUB    .S1     A1,1,A1       ;decrement count
;cycle 5
        ||    LDDW     .D1     *A4++,A3:A2   ;64-bit data in A2 and A3
        ||    LDDW     .D2     *B4++,B3:B2   ;64-bit data in B2 and B3
        || [A1] SUB    .S1     A1,1,A1       ;decrement count
        || [A1] B      .S2     LOOP          ;branch to LOOP
```

```
;cycle 6
        ||        LDDW     .D1     *A4++,A3:A2    ;64-bit data in A2 and A3
        ||        LDDW     .D2     *B4++,B3:B2    ;64-bit data in B2 and B3
        || [A1]   SUB      .S1     A1,1,A1        ;decrement count
        || [A1]   B        .S2     LOOP           ;branch to LOOP
        ||        MPYSP    .M1x    A2,B2,A6       ;lower 32-bit product into A6
        ||        MPYSP    .M2x    B3,A3,B6       ;upper 32-bit product into B6
;cycle 7
        ||        LDDW     .D1     *A4++,A3:A2    ;32-bit data in A2 and A3
        ||        LDDW     .D2     *B4++,B3:B2    ;32-bit data in B2 and B3
        || [A1]   SUB      .S1     A1,1,A1        ;decrement count
        || [A1]   B        .S2     LOOP           ;branch to LOOP
        ||        MPYSP    .M1x    A2,B2,A6       ;lower 32-bit product into A6
        ||        MPYSP    .M2x    B3,A3,B6       ;upper 32-bit product into B6
;cycle 8
        ||        LDDW     .D1     *A4++,A3:A2    ;32-bit data in A2 and A3
        ||        LDDW     .D2     *B4++,B3:B2    ;32-bit data in B2 and B3
        || [A1]   SUB      .S1     A1,1,A1        ;decrement count
        || [A1]   B        .S2     LOOP           ;branch to LOOP
        ||        MPYSP    .M1x    A2,B2,A6       ;lower 32-bit product into A6
        ||        MPYSP    .M2x    B3,A3,B6       ;upper 32-bit product into B6
;cycle 9
        ||        LDDW     .D1     *A4++,A3:A2    ;32-bit data in A2 and A3
        ||        LDDW     .D2     *B4++,B3:B2    ;32-bit data in B2 and B3
        || [A1]   SUB      .S1     A1,1,A1        ;decrement count
        || [A1]   B        .S2     LOOP           ;branch to LOOP
        ||        MPYSP    .M1x    A2,B2,A6       ;lower 32-bit product into A6
        ||        MPYSP    .M2x    B3,A3,B6       ;upper 32-bit product into B6

;cycles 10-109 (loop kernel)
        ||        LDDW     .D1     *A4++,A3:A2    ;32-bit data in A2 and A3
        ||        LDDW     .D2     *B4++,B3:B2    ;32-bit data in B2 and B3
        || [A1]   SUB      .S1     A1,1,A1        ;decrement count
        || [A1]   B        .S2     LOOP           ;branch to LOOP
        ||        MPYSP    .M1x    A2,B2,A6       ;lower 32-bit product into A6
        ||        MPYSP    .M2x    B3,A3,B6       ;upper 32-bit product into B6
        ||        ADDSP    .L1     A6,A7,A7       ;accum in A7
        ||        ADDSP    .L2     B6,B7,B7       ;accum in B7
;branch occurs here
;cycles 110-124 (epilog)
                  ADDSP    .L1x    A7,B7,A0       ;lower/upper sum of products
                  ADDSP    .L2x    A7;B7,B0       ;
                  ADDSP    .L1x    A7,B7,A0       ;
                  ADDSP    .L2x    A7,B7,B0       ;
                  NOP                             ;wait for 1st B0
                  ADDSP    .L1x    A0,B0,A5       ;1st two sum of products
                  NOP                             ;wait for 2nd B0
                  ADDSP    .L2x    A0,B0,B5       ;last two sum of products
                  NOP              3              ;3 delay slots for ADDSP
                  ADDSP    .L1x    A5,B5,A4       ;final sum
                  NOP              3              ;3 delay slots for final sum
```

图 8.14 利用软件流水线方法，浮点实现点积的汇编程序（dotpipedfloat.asm）

循环周期	指令	
110	ADDSP	
111	ADDSP	
112	ADDSP	
113	ADDSP	
114	NOP	
115	ADDSP	
116	NOP	
117	ADDSP	
118-120	NOP	3
121	ADDSP	
122-124	NOP	3

从时钟周期113到116，A7内保存低32位乘积的和，而B7内保存高32位乘积的和，即：

循环周期	A7（包含低32位）和B7（包含高32位）
113	p0 + p4 + p8 + ... + p96
114	p1 + p5 + p9 + ... + p97
115	p2 + p6 + p10 + ... + p98
116	p3 + p7 + p11 + ... + p99

在第114个时钟周期，可得到A0 = A7 + B7的值。A0累加低位和高位部分的乘积的和。这里：

```
A7 = p0 + p4 + p8 +···+ p96     (低32位)
B7 = p0 + p4 + p8 +···+ p96     (高32位)
```

在第115个时钟周期，可得到B0 = A7 + B7的值，这里：

```
A7 = p1 + p5 + p9 +···+ p97     (低32位)
B7 = p1 + p5 + p9 +···+ p97     (高32位)
```

类似地，在第116个和第117个时钟周期，可得到A0和B0的值如下：

```
A0 = (p2 + p6 + p10 +···+ p98)    高低32位的和
B0 = (p3 + p7 + p11 +···+ p99)    高低32位的和
```

在第119个时钟周期，可得到A5 = A0 + B0（从第114个和第115个时钟周期得到）的值；在第121个时钟周期，可得到B5 = A0 + B0（从第116个和第117个时钟周期得到）的值。

最终的和在寄存器A4累加，并在第124个时钟周期后可得到求和的结果。

8.6　不同优化方案执行的时钟周期比较

表8.14给出了不同优化方法所需的时钟周期情况，包括定点实现和浮点实现方式，这里count = 200。对于不同大小的数组，由于开始部分和结尾部分包含的时钟周期数是相同的，因此总的时钟周期数由数组的大小决定。

注意，如果 count = 1000，利用软件流水线方法，定点实现和浮点实现所需的时钟周期数如下。

定点：7 + (count/2) + 1 = 508 个时钟周期

浮点：9 + (count/2) + 15 = 524 个时钟周期

表 8.4　不同优化方法定点和浮点实现所需要的时钟周期数（count = 200）

优化方案	周期数	
	定　点	浮　点
不优化	2 + (16 × 200) = 3202	2 + (18 × 200) = 3602
使用并行指令	1 + (8 × 200) = 1601	1 + (10 × 200) = 2001
每次迭代时两个求和	1 + (8 × 100) = 801	1 + (10 × 100) + 7 = 1008
使用软件流水线操作	7 + (100) + 1 = 108	9 + (100) + 15 = 124

参考文献

1. *TMS320C6000 Programmer's Guide,* SPRU198D, Texas Instruments, Dallas, TX, 2000.
2. *Guidelines for Software Development Efficiency on the TMS320C6000 VelociTI Architecture,* SPRA434, Texas Instruments, Dallas, TX, 1998.
3. *TMS320C6000 CPU and Instruction Set,* SPRU189F, Texas Instruments, Dallas, TX, 2000.
4. *TMS320C6000 Assembly Language Tools User's Guide,* SPRU186G, Texas Instruments, Dallas, TX, 2000.
5. *TMS320C6000 Optimizing Compiler User's Guide,* SPRU 187G, Texas Instruments, Dallas, TX, 2000.

第 9 章　DSP 的应用及学生的课题

这一章可当做实验、学生的课题和 DSP 应用的资料，也可以作为文献[1-4]的参考。C30 和 C31 浮点处理器[5-20]及 TMS320C25 定点处理器[21-26]已广泛应用于多种工程课题中，涉及通信、控制及神经网络等应用领域，这些课题可作为其他工程应用的参考。TI 公司每年出版的论文集中，包含了许多基于 TMS320 系列数字信号处理器的论文，这些论文可为其他工程项目提供很好的借鉴。TI 公司网站上有一系列学生的课程设计，内容涵盖的应用范围很宽，这些课题已成为 TI 公司“DSP 和模拟设计挑战赛”的决赛内容（一等奖奖金为 100 000 美元），另外，第 6 章、第 7 章和附录 D 至附录 F 的内容也是非常有用的。

感谢所有为写这一章提供素材的学生，包括来自 Roger Williams 大学和 Massachusetts-Dartmouth 大学的学生，他们为我的 DSP 应用内容提供了大部分的应用背景，尤其是参加我的研究生课程——基于 C6x 的实时数字信号处理器的学习，来自 Worcester Polytechnic Institute（WPI）的学生：Y. Bognadov, J. Boucher, G. Bowers, D. Ciota, P. DeBonte, B. Greenlaw, S. Kintigh, R. Lara-Montalvo, M. Mellor, F. Moyse, A. Pandey, I . Progri, V. C. Ramanna, P. Srikrishna, U. Ummethala 和 L. Wan。本章对他们的课题（以及一些小课题）进行了简要的讨论。第 6 章和第 7 章讨论了自适应滤波和图像均衡器两个课题。

9.1　使用 DMA 和用户开关的话音扰乱器

工程 scram16k_sw（在辅助材料中）是例 4.9 的扩展，它使用了三个拨码（DIP）开关：USER_SW1 到 USER_SW3，DSK 板上有 4 个这样的开关（而第四个开关没有使用）。当语音信号输入时，根据用户开关的设置，输出可以是非扰乱的语音信号。

用户的 DIP 开关用于决定是否上抽样。根据开关的位置，程序可以作为环路或滤波程序。USER_SW1 对应最低有效位（LSB），如程序中第一个测试的设置开关设置为“下/下/上”，表示 $(001)_b$。如果测试条件为真，输出是 16 kHz 上抽样的话音扰乱信号。下面是开关位置设置与功能的关系：

	USER_SW1	USER_SW2	USER_SW3	
a.	0	0	1	输出 $F_s = 16$ kHz 的扰乱语音信号
b.	1	0	1	输出 $F_s = 16$ kHz 的未扰乱的语音信号
c.	1	1	1	$F_s = 16$ kHz 的低通滤波
d.	0	1	0	输出 $F_s = 8$ kHz 的扰乱语音信号
e.	1	1	0	输出 $F_s = 8$ kHz 的未扰乱语音信号
f.	0	0	0	$F_s = 8$ kHz 的低通滤波
g.	1	0	0	环路程序

scram8k_DMA

另一个替代的课题 scram8k_DMA（在辅助材料中）使用了 DMA 方法和 8 kHz 的抽样频率，实现了语音信号的扰乱。它是由 DSK 工具包中 codec_edma 的例子经过修改而成的，该程序演示了使用 DMA 以及选择不同设置，实现环路程序、滤波或语音扰乱（没有上抽样）等不同的功能。

9.2　锁相环

锁相环课题实现了基于软件的线性锁相环路（PLL），基本锁相环使一个特定的系统去跟踪另一个锁相环。锁相环包括相位检测器、环路滤波器和压控振荡器。软件锁相环更为通用，功能也更强大，但覆盖的频率范围有限，这是由于在每个输入信号周期，输入信号必须至少执行一次锁相环的功能[27-29]，也就是要进行相应的运算。

最初，锁相环是用 MATLAB 编写测试的，然后再使用 C 编程移植到 C6x 上。锁相环锁定到由程序内部或者外部信号源产生的正弦波上，输出信号显示在示波器上或使用 DSP/BIOS 的实时数据传输（RTDX）工具传输到 PC 上进行观察。

图 9.1 给出了两种形式线性锁相环的原理框图。

1. 使用外部输入信号源，数控振荡器（DCO）的输出作为示波器的输入；
2. 由查表方法产生正弦波，使用 RTDX 工具以及用 Excel 观察不同信号。

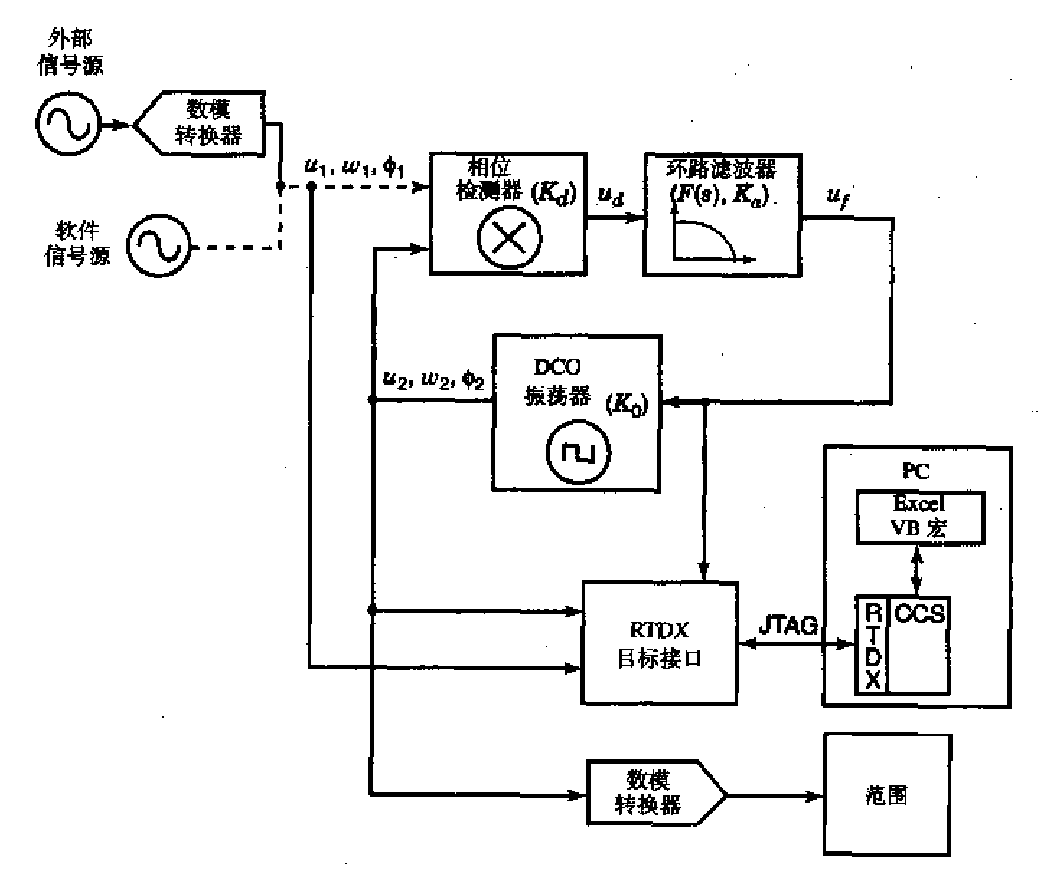


图 9.1　PLL 的原理框图

图 9.1 的相位检测器将输入的正弦信号与 DCO 输出的方波信号相乘，得到两个输出信号，一个信号频率是两个输入信号频率的和，另一个是两个输入信号频率的差，前者对应的是高频输出

分量，后者对应的是低频输出分量。低频分量用于控制环路，而高频分量则被环路滤波器滤除。当锁相环路锁定时，相位检测器的两个输入信号是同频率的正交（90度）信号。

环路滤波器是一个低通滤波器，它允许相位检测器输出的低频分量通过，抑制不需要的高频分量。环路滤波器是单零点单极点的 IIR 滤波器，零点的作用是提高环路的动态特性和稳定性。环路滤波器定标的输出表示 DCO 相位的瞬时增量。DCO 输出方波可以是 Walsh 函数：+1 表示相位在 0~π 之间；而−1 表示相位在−π~0 之间，相位增量与输入的数据成比例。

9.2.1　RTDX 用于实时数据传输工具

RTDX 用于向 PC 传输数据，而 PC 用查表方法生成的正弦波作为输入信号。输出通道把输入信号、环路滤波器和 DCO 的输出信号以及时间戳传输到 CCS。CCS 缓冲这些数据，以便其他应用程序能读取这些数据。CCS 有一个接口，使 PC 应用程序能访问缓冲的 RTDX 数据。Visual Basic Excel（LABVIEW 和 Visual C++也可以）用于在 PC 显示器上显示结果。

9.3　SB-ADPCM 编解码器：G.722 语音编码器的实现

音频信号以 16 kHz 的速率进行抽样，数据传输速率为 64 kbits/s，在接收端重建音频信号[30,31]。

编码器

子带自适应差分脉冲编码调制器（SB-ADPCM）由正交镜像发送滤波器组成，它将输入信号分成一个 0~4 kHz 的低频信号和一个 4~8 kHz 的高频信号。低频和高频信号分别对各自自适应预测器输出的误差信号进行动态量化和编码，两个误差信号分别用 6 位和 2 位进行编码。只要误差信号足够小，总的量化噪声就可以忽略，系统性能就能保证较好。将低频和高频两部分的编码信号复接后，就变成 8 kHz 抽样率 8 位的信号，这样数据速率就变成 64 kbits/s。图 9.2 给出了 SB-ADPCM 编码器的原理框图。

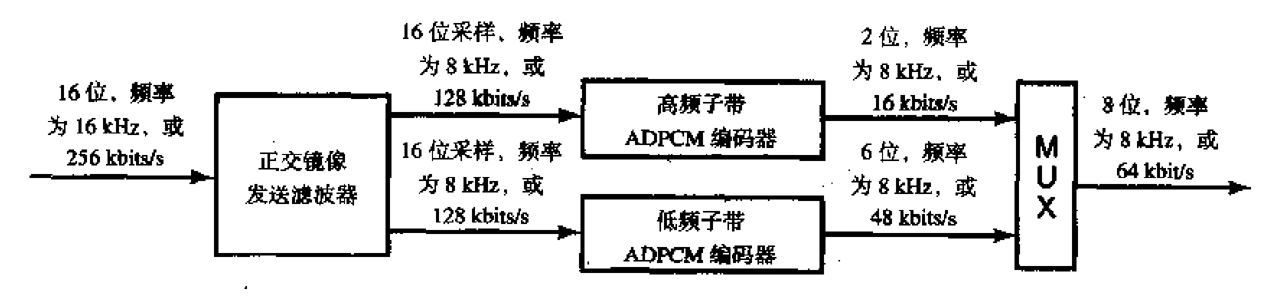


图 9.2　ADPCM 编码器的原理框图

正交镜像发送滤波器

正交镜像发送滤波器（QMF）接收速率为 16 kHz 的 16 位抽样，然后把它分成低频和高频信号两部分。滤波器系数表示 4 kHz 的低通滤波器。抽样信号分成奇抽样和偶抽样，频率在 4~8 kHz 之间的信号有混叠效应。这种混叠效应使高频奇抽样和高频偶抽样的相位差为 180 度，而低频的奇、偶抽样是同相的。当奇偶信号相加时，经过滤波器后低频信号之间奇偶抽样相加，而高频信号之间奇偶抽样相互抵销，生成抽样率为 8 kHz 的低频信号。

低频子带编码器将从 QMF 输出的低频信号转换成一个误差信号，并量化成 6 位的数字信号。

解码器

如图 9.3 所示，解码器将 64 kbits/s 信号分成两部分信号，分别输入到低频和高频的子带 SB-ADPCM 解码器。正交镜像接收滤波器（QMRF）由两个数字滤波器组成，对低频和高频子带 ADPCM 解码器的 8~16 kHz 信号进行插值，生成速率为 16 kHz 的信号。在高频 SB-ADPCM 解码器中，将量化误差信号与估计的信号相加来重构信号。

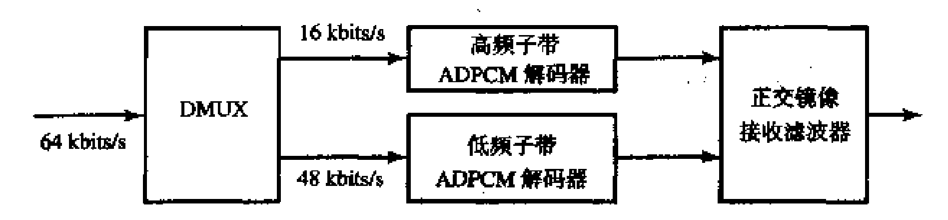


图 9.3　ADPCM 解码器的原理框图

ADPCM 解码器的解码过程包括逆自适应量化器、量化器调整、自适应预测、预测值计算和重构信号计算。用 CD 播放器播放的信号作为输入，可以听出 DSK 重构的输出信号音质是较好的。通过比较缓冲的输入数据和重构的输出数据，也可以说明解码器的结果是较为理想的。

9.4　自适应时域衰减器

自适应时域衰减器（ATA）抑制不需要的窄带信号，以获取最大的信干比。图 9.4 是自适应时域衰减器的原理框图。输入通过延时单元，从选择的延迟单元输出的信号经过系数加权，输出为：

$$y[k]=\mathbf{m}^T\cdot\mathbf{r}[k]=\sum_{i=0}^{N-1}(\mathbf{m}_i\cdot\mathbf{r}[k-i])$$

其中$\mathbf{m}$是权重系数矢量，$\mathbf{r}$是从输入信号中选出的延迟抽样矢量，N是$\mathbf{m}$和$\mathbf{r}$中抽样点的数目。自适应算法基于相关矩阵和方向矢量计算加权系数：

$$\mathbf{C}[k,\delta=0]\cdot\mathbf{m}[k]=\lambda\mathbf{D}$$

其中$\mathbf{C}$是相关矩阵，$\mathbf{D}$是方向矢量，λ是比例因子。相关矩阵$\mathbf{C}$是在几个抽样周期的信号相关的均值：

$$\mathbf{C}[k,\delta]=\frac{1}{N_{\mathrm{AV}}}\sum_{i=0}^{n-1}\left(\mathbf{r}[k]\otimes\mathbf{r}[k-\delta]^T\right)$$

这里N_{AV}是在均值计算中样本的数目。方向矢量$\mathbf{D}$表示期望的信号：

$$\mathbf{D}=[1\quad\exp(j\omega_T\tau)\quad\cdots\quad\exp(j\omega_T(N-1)\tau)]^T$$

ω_T是期望信号的角频率，τ是输出样本之间的延时，N是相关矩阵的阶数。

该过程使干扰信号与期望的信号之比（UDR）最小[32]，UDR 定义为信号总功率与期望信号功率之比，也就是：

$$\mathrm{UDR}=\frac{P_{\mathrm{total}}}{P_d}=\frac{\mathbf{m}[k]^T\cdot\mathbf{C}[k,0]\cdot\mathbf{m}[k]}{P_d\left(\mathbf{m}[k]^T\cdot\mathbf{D}\right)^2}=\frac{1}{P_d\left(\mathbf{m}[k]^T\cdot\mathbf{D}\right)}$$

其中P_d是期望信号的功率。

用 MATLAB 对自适应时域衰减器进行仿真，然后再把它传输到 C6x 中实时实现。图 9.5 给

出了使用频率为 1416 Hz 期望信号和 1784 Hz 干扰信号（可以改为其他频率信号）的测试配置框图。用 MATLAB 找到能使 UDR 最小化的最佳τ值，因为使用该τ值（随 GEL 文件而改变）可使干扰信号（最初显示在 HP3561A 分析仪上）极大地衰减。关于这一点，在实时处理中也得到了验证。

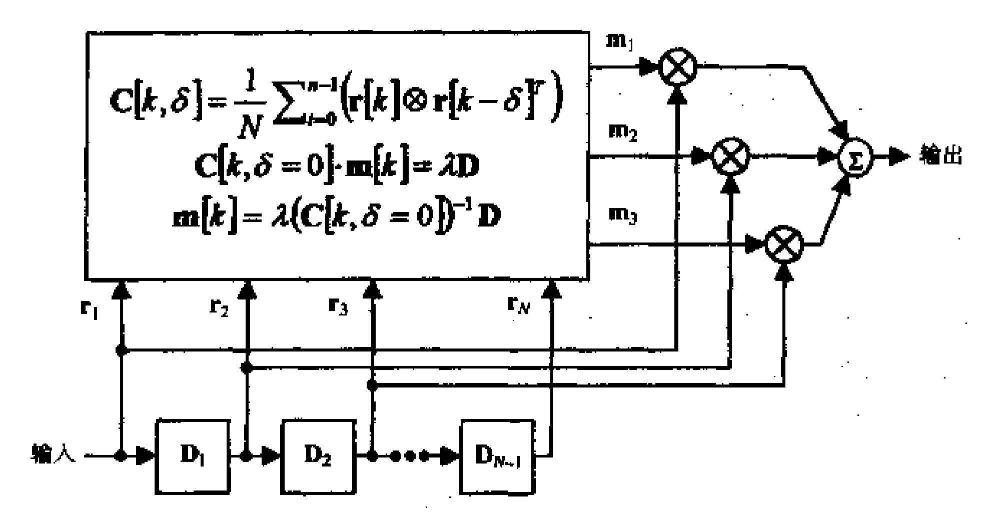


图 9.4　自适应时域衰减器原理框图

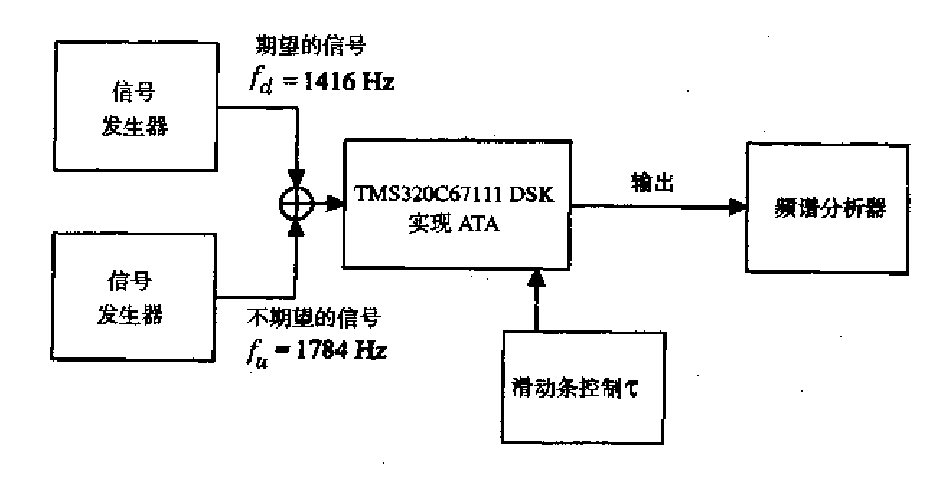


图 9.5　自适应时域衰减器的测试配置图

9.5　图像处理

本课题实现了图像处理中使用的多种方案：

1．边缘检测，使用 Sobe 的边缘检测来增强图像的边缘。
2．中值滤波，非线性滤波器滤除图像中的尖峰噪声。
3．直方图均衡，利用图像谱。
4．图像模糊遮蔽，空间滤波器锐化图像，增强图像的高频分量。
5．点检测，增强图像中单点特点。

主要问题是：（由于学期课程时间的限制），使用/调用像.h 的图像文件，而不是使用实时图像。在该课题期间，同时开发了下面一些算法程序：均值、方差可调的加性高斯噪声的程序实例以及将某种概率分布的一组数据映射为另一种不同分布的直方图变换程序实例（用于图像处理中）。

9.6 用改进的 Prony 方法设计和实现滤波器

本课题是基于改进的 Prony 方法来设计和实现滤波器的[33-36]。这种方法基于滤波器表示的相关性，不需要任何微分计算或系数矢量初始估值，利用递推的方法计算滤波器的系数，得到滤波器的冲激响应。

9.7 FSK 调制解调器

本课题实现了一个数字调制解调器，它产生 8 进制的 FSK 载波信号。程序执行步骤如下：

1. 把采集的数据作为输入信号；
2. 将 6 个最重要的比特分成两个 3 比特的抽样；
3. 抽样数据最重要的 3 比特选择一个频率的 FSK 载波信号；
4. FSK 载波送给解调器；
5. FSK 载波加 Hanning 窗函数；
6. 对加窗的 FSK 载波信号进行（16 点）DFT 运算；
7. DFT 运算的结果送给频率选择函数，它选择幅度最高的频率，而该频率对应抽样数据的高 3 位；
8. 对抽样数据的次重要 3 比特重复上述过程；
9. 高和低 3 比特合并送给编码/解码器；
10. GEL 程序有一个选项，用来插值或上抽样重构的数据以得到更平滑的输出波形。

9.8 μ律语音压扩

诸如将语音这样的模拟输入信号转换成数字信号并压缩成 8 比特数据，μ律编码就是一种针对语音信号进行对数压缩非均匀量化的方案。在美国，语音信号进行编码传输时，先采用μ律对信号进行对数形式的压缩，以便在不提高数据量的前提下提高信噪比，这种方法现在在电信领域被广泛应用。

尽管量化的位数保持不变，但动态范围增加了。μ律压缩的语音信号一般用 8 比特抽样数据表示，它携带小信号的信息量要比大信号的信息量多。采用压缩技术基于这样的事实：从统计意义上来讲，信号更可能在小信号区间，而不是在大信号区间，因此，在小信号区间需要更多的量化点数。

查找表内共有 256 个数[①]，它们分别用来获得 0 到 7 段的量化电平，该表由 16×16 组数组成：

0 段间隔为 2
1 段间隔为 2
2 段间隔为 4
3 段间隔为 8
4 段间隔为 16
5 段间隔为 32
6 段间隔为 64
7 段间隔为 128

① 译者注：书中这段描述与给出的表没有对应关系，可参照 PCM 编码原理理解该段意思。

大信号多数是由第 7 段来表示的（从表上看出）。3 个指数位用来表示第 0 段到第 7 段，4 个尾数位用于表示后 4 个有效位，还有一位是符号位。

16 比特的输入数据是由线性数据变换成 8 比特μ律数据（模拟传输），然后再从μ律转换成 16 比特的线性数据（模拟接收），再输出到编解码器。

9.9　语音检测及逆回放

本课题用话筒检测语音信号，然后再从相反方向回放。课题使用了两个缓冲区：输入缓冲区可保存不断更新的 80 000 个抽样（10 秒的数据量），输出缓冲区用来反向回放输入的语音信号。课题能检测信号的电平并跟踪包络的变化，从而决定是否有语音信号。

当语音信号出现，然后又逐渐消失时，信号电平检测器发出命令，启动信号回放过程，这时存储的数据由输入缓冲区传输到回放的输出缓冲区。当到达整个检测到的信号的结尾时，停止回放过程。

信号电平检测方案包括整流和滤波（用简单一阶 IIR 滤波器实现）。当信号到达较高的门限电平时，显示器就会有相应的显示；当信号下降到较低的门限时，就可以计算语音开始和结束的时间差。如果该时间差小于指定的时间间隔，程序就继续进入无信号状态（假设只有噪声），否则，如果该时间差比指定的时间间隔长，这时就会激活信号检测模式。

9.10　其他课题

下面介绍一些其他课题，它们使用 C/C3x 和 C2x/C5x 程序或指令实时实现了课题。

9.10.1　声波方向跟踪器

文献[15]讨论了声波方向跟踪器，并用 C/C3x 程序指令实现了它。使用两个话筒来捕获信号，根据信号到达两个话筒的时延，可确定声源位置的相对角度。信号从源辐射一定距离后，可认为有如图 9.6 所示平面波前，这样就可以把传感器（许多话筒可作为声学传感器）等间隔地排成一行，确定信号辐射的角度。如果一个话筒比另一个更靠近信号源，较远的话筒接收到的信号就有较大的时间延迟，该时间延迟与信号源位置的角度以及话筒和信号源之间的相对距离相对应。角度 $c = \arcsin(a/b)$，这里距离 a 是声速和延迟时间的乘积（相位/频率）。

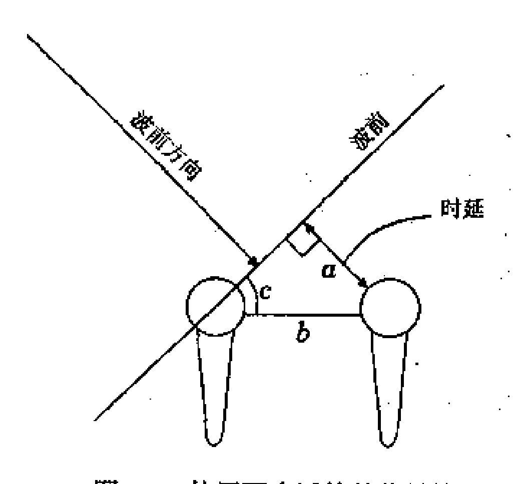


图 9.6　使用两个话筒的信号接收

图 9.7 所示的框图是声音信号的跟踪器。通过它可得到两个 128 点的数组数据，计算第一个信号与第二个信号的互相关，再将互相关的结果数据分解成两部分，每次变换使用 128 点的 FFT，最后的相位是两个信号的相位差。

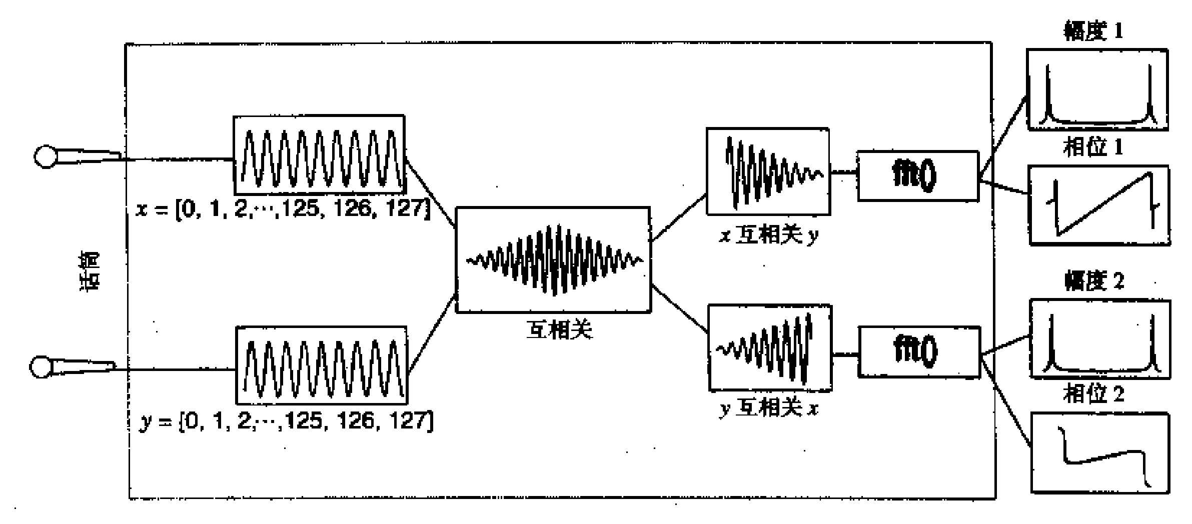


图 9.7　声音信号跟踪器的框图

9.10.2　多速率滤波器

使用多速率处理方法，可用较少的系数实现滤波器，而使用等效的单一速率方法却不然。文献[37-44]讨论并利用 C3x/C4x 和 C2x/C5x 兼容指令实现了多速率滤波器。多速率滤波器潜在的应用包括图形均衡器、受控噪声源、背景噪声合成器。多速率滤波处理使用多个抽样速率实现需要的处理操作。两个基本的处理是抽取和插值，前者是减小抽样速率，后者是提高抽样速率[38-42]。多速率抽取器能减小滤波器的运算量。抽样速率增加 K 倍可通过在 X_i 和 X_{i+1} 之间顺序插入 $K-1$ 个 0 来实现，多级抽取和插值一般具有较高的效率。

二进制随机信号输入到一组滤波器，形成输出信号的功率谱，图 9.8 给出了 10 种带宽的多速率滤波器功能原理框图。关于该滤波器已经进行了讨论，并利用 C3x[37]和 C2x/C5x 指令[43,44]实现了它，其频率范围分为 10 倍频程频带，每个频带的 1/3 倍频程是可调节的。

9.10.3　神经网络在信号识别中的应用

将信号经过 FFT 处理的结果输入到神经网络，经过训练的神经网络，该网络是利用 C 语言实现的，并采用后向传播学习算法[45,46]识别输入信号。为了说明该算法，图 9.9 给出了 7 节点的三层神经网络模型。现在有许多不同的神经网络训练算法，其中后向传播算法已经得到广泛的应用。假设输入一组信号，训练网络给出要求的响应。如果神经网络给出错误的输出时，网络就需要调整参数（权系数），从而使网络能够得到修正，减少错误的发生。在修正调整过程中，从输出节点开始，向后传播到输入节点。

9.10.4　PID 控制器

在文献[6,47,48]中，使用了比例、积分和微分（PID）控制算法，实现了自适应和非自适应控制器。

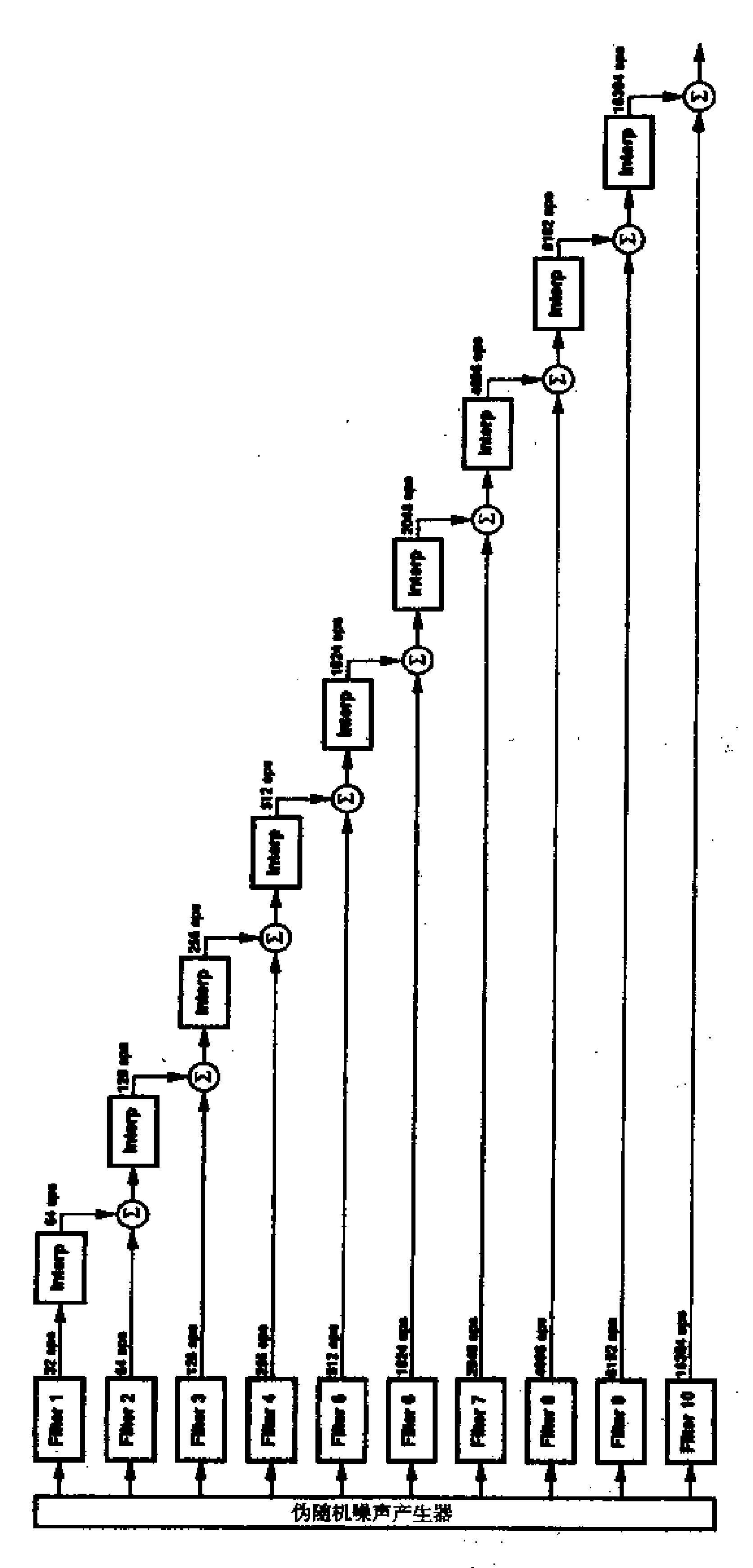


图 9.8 10 种带宽的多速率滤波器功能框图

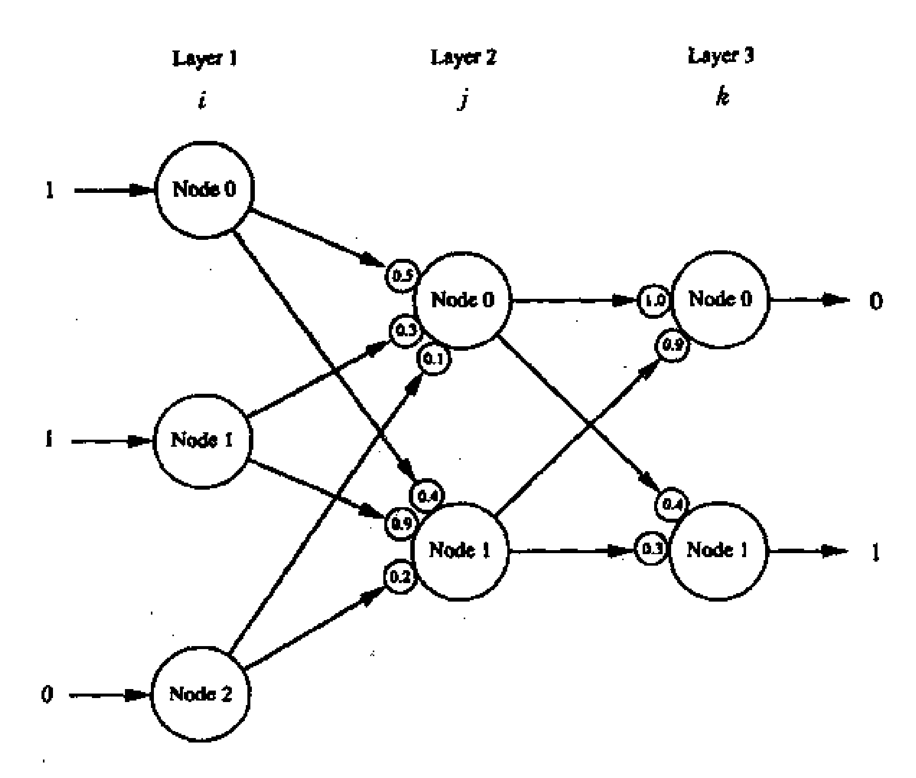


图 9.9 7 个节点的三层神经网络

9.10.5 用于快速获得数据的四通道复用器

为了完成该课题，用 C 程序设计和搭建了一个四通道复用器模块[6]。课题使用一个 8 位闪烁（Flash）ADC，一个 FIFO，一个 MUX，一个晶振（2 MHz 或者 20 MHz）。使用 4 个通道中的一个通道对一路输入信号进行采集，输入信号的 FFT 实时显示在 PC 显示器上。

9.10.6 视频行速率分析

该课题在文献[6,49]中进行了讨论，并用 C/C3x 实现了行速率视频信号的分析。交互式算法通常用于图像处理中的滤波、平均和边缘锐化，这些算法是用 C 语言编写的，实现了图像的分析。用 CCD 摄像机作为视频信号源，输入给该课题设计的一个电路模块，该模块包括触发器、逻辑门和时钟。PC 显示器上显示了使用 500 kHz 或 3 MHz 的 IIR 低通滤波器以及边缘锐化算法对视频行信号的不同影响。

参考文献

1. J. H. McClellan, R. W. Schafer, and M. A. Yoder, *DSP First: A Multimedia Approach*, Prentice Hall, Upper Saddle River, NJ, 1998.
2. N. Kehtarnavaz and M. Keramat, *DSP System Design Using the TMS320C6000*, Prentice Hall, Upper Saddle River, NJ, 2001.
3. N. Dahnoun, *DSP Implementation Using the TMS320C6x Processors*, Prentice Hall, Upper Saddle River, NJ, 2000.
4. M. Morrow, T. Welch, C. Cameron, and G. York, Teaching real-time beamforming with the C6211 DSK and MATLAB, *Proceedings of the Texas Instruments DSPS Fest Annual*

Conference, 2000.

5. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, Wiley, New York, 1999.
6. R. Chassaing, *Digital Signal Processing with C and the TMS320C30*, Wiley, New York, 1992.
7. C. Marven and G. Ewers, *A Simple Approach to Digital Signal Processing*, Wiley, New York, 1996.
8. J. Chen and H. V. Sorensen, *A Digital Signal Processing Laboratory Using the TMS320C30*, Prentice Hall, Upper Saddle River, NJ, 1997.
9. S. A. Tretter, *Communication System Design Using DSP Algorithms*, Plenum Press, New York, 1995.
10. R. Chassaing et al., Student projects on digital signal processing with the TMS320C30, *Proceedings of the 1995 ASEE Annual Conference*, June 1995.
11. J. Tang, Real-time noise reduction using the TMS320C31 digital signal processing starter kit, *Proceedings of the 2000 ASEE Annual Conference*, 2000.
12. C. Wright, T. Welch III, M. Morrow, and W. J. Gomes III, Teaching real-world DSP using MATLAB and the TMS320C31 DSK, *Proceedings of the 1999 ASEE Annual Conference*, 1999.
13. J. W. Goode and S. A. McClellan, Real-time demonstrations of quantization and prediction using the C31 DSK, *Proceedings of the 1998 ASEE Annual Conference*, 1998.
14. R. Chassaing and B. Bitler (contributors), Signal processing chips and applications, *The Electrical Engineering Handbook*, CRC Press, Boca Raton, FL, 1997.
15. R. Chassaing et al., Digital signal processing with C and the TMS320C30: Senior projects, *Proceedings of the 3rd Annual TMS320 Educators Conference*, Texas Instruments, Dallas, TX, 1993.
16. R. Chassaing et al., Student projects on applications in digital signal processing with C and the TMS320C30, *Proceedings of the 2nd Annual TMS320 Educators Conference*, Texas Instruments, Dallas, TX, 1992.
17. R. Chassaing, TMS320 in a digital signal processing lab, *Proceedings of the TMS320 Educators Conference*, Texas Instruments, Dallas, TX, 1991.
18. P. Papamichalis, ed., *Digital Signal Processing Applications with the TMS320 Family: Theory, Algorithms, and Implementations*, Vols. 2 and 3, Texas Instruments, Dallas, TX, 1989 and 1990.
19. *Digital Signal Processing Applications with the TMS320C30 Evaluation Module: Selected Application Notes*, Texas Instruments, Dallas, TX, 1991.
20. R. Chassaing and D. W. Horning, *Digital Signal Processing with the TMS320C25*, Wiley, New York, 1990.
21. I. Ahmed, ed., *Digital Control Applications with the TMS320 Family*, Texas Instruments, Dallas, TX, 1991.
22. A. Bateman and W. Yates, *Digital Signal Processing Design*, Computer Science Press, New York, 1991.
23. Y. Dote, *Servo Motor and Motion Control Using Digital Signal Processors*, Prentice Hall, Upper Saddle River, NJ, 1990.
24. R. Chassaing, A senior project course in digital signal processing with the TMS320, *IEEE Transactions on Education*, Vol. 32, 1989, pp. 139–145.

25. R. Chassaing, Applications in digital signal processing with the TMS320 digital signal processor in an undergraduate laboratory, *Proceedings of the 1987 ASEE Annual Conference*, June 1987.

26. K. S. Lin, ed., *Digital Signal Processing Applications with the TMS320 Family: Theory, Algorithms, and Implementations*, Prentice Hall, Upper Saddle River, NJ, Vol. 1, 1988.

27. Roland E. Best, *Phase-Locked Loops Design, Simulation, and Applications*, 4th ed., McGraw-Hill, New York, 1999.

28. W. Li and J. Meiners, *Introduction to Phase-Locked Loop System Modeling*, SLTT015, Texas Instruments, Dallas, TX, May 2000.

29. J. P. Hein and J. W. Scott, Z-domain model for discrete-time PLL's, *IEEE Transactions on Circuits and Systems*, Vol. CS-35, Nov. 1988, pp. 1393–1400.

30. *ITU-T Recommendation G.722 Audio Coding with 64 kbits/s.*

31. P. M. Embree, *C Algorithms for Real-Time DSP*, Prentice Hall, Upper Saddle River, NJ, 1995.

32. I. Progri and W. R. Michalson, Adaptive spatial and temporal selective attenuator in the presence of mutual coupling and channel errors, *ION GPS-2000.*

33. F. Brophy and A. C. Salazar, Recursive digital filter synthesis in the time domain, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-22, 1974.

34. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, New York, 1992.

35. J. Borish and J. B. Angell, An efficient algorithm for measuring the impulse response using pseudorandom noise, *Journal of the Audio Engineering Society*, Vol. 31, 1983.

36. T. W. Parks and C. S. Burrus, *Digital Filter Design*, Wiley, New York, 1987.

37. R. Chassaing, P. Martin, and R. Thayer, Multirate filtering using the TMS320C30 floating-point digital signal processor, *Proceedings of the 1991 ASEE Annual Conference*, June 1991.

38. R. E. Crochiere and L. R. Rabiner, *Multirate Digital Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1983.

39. R. W. Schafer and L. R. Rabiner, A digital signal processing approach to interpolation, *Proceedings of the IEEE*, Vol. 61, 1973, pp. 692–702.

40. R. E. Crochiere and L. R. Rabiner, Optimum FIR digital filter implementations for decimation, interpolation and narrow-band filtering, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-23, 1975, pp. 444–456.

41. R. E. Crochiere and L. R. Rabiner, Further considerations in the design of decimators and interpolators, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-24, 1976, pp. 296–311.

42. M. G. Bellanger, J. L. Daguet, and G. P. Lepagnol, Interpolation, extrapolation, and reduction of computation speed in digital filters, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-22, 1974, pp. 231–235.

43. R. Chassaing, W. A. Peterson, and D. W. Horning, A TMS320C25-based multirate filter, *IEEE Micro*, Oct. 1990, pp. 54–62.

44. R. Chassaing, Digital broadband noise synthesis by multirate filtering using the TMS320C25, *Proceedings of the 1988 ASEE Annual Conference*, Vol. 1, June 1988.

45. B. Widrow and R. Winter, Neural nets for adaptive filtering and adaptive pattern recognition, *Computer*, Mar. 1988, pp. 25–39.

46. D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, MIT Press, Cambridge, MA, 1986.

47. J. Tang, R. Chassaing, and W. J. Gomes III, Real-time adaptive PID controller using the TMS320C31 DSK *Proceedings of the 2000 Texas Instruments DSPS Fest Conference*, 2000.

48. J. Tang and R. Chassaing, PID controller using the TMS320C31 DSK for real-time motor control, *Proceedings of the 1999 Texas Instruments DSPS Fest Conference*, 1999.

49. B. Bitler and R. Chassaing, Video line rate processing with the TMS320C30, *Proceedings of the 1992 International Conference on Signal Processing Applications and Technology (ICSPAT)*, 1992.

50. *MATLAB, The Language of Technical Computing, Version 6.3*, MathWorks, Natick, MA, 1999.

附录 A　TMS320C6x 指令集

A.1　定点和浮点操作指令

表 A.1 列出了 C6x 处理器的指令，表中的这些指令是按照它们所使用的功能单元来进行分类的，这些指令可用于 C6x 的定点和浮点处理器。

表 A.1　定点和浮点操作指令

.L Unit	.M Unit	.S Unit	.D Unit
ABS	MPY	ADD	ADD
ADD	MPYH	ADDK	ADDAB
ADDU	MPYHL	ADD2	ADDAH
AND	MPYHLU	AND	ADDAW
CMPEQ	MPYHSLU	B disp	LDB
CMPGT	MPYHSU	B IRP*	LDBU
CMPGTU	MPYHU	B NRP*	LDH
CMPLT	MPYHULS	B reg	LDHU
CMPLTU	MPYHUS	CLR	LDW
LMBD	MPYLH	EXT	LDB (15-bit offset)**
MV	MPYLHU	EXTU	LDBU (15-bit offset)**
NEG	MPYLSHU	MV	LDH (15-bit offset)**
NORM	MPYLUHS	MVC*	LDHU (15-bit offset)**
NOT	MPYSU	MVK	LDW (15-bit offset)**
OR	MPYU	MVKH	MV
SADD	MPYUS	MVKLH	STB
SAT	SMPY	NEG	STH
SSUB	SMPYH	NOT	STW
SUB	SMPYHL	OR	STB (15-bit offset)**
SUBU	SMPYLH	SET	STH (15-bit offset)**
SUBC		SHL	STW (15-bit offset)**
XOR		SHR	SUB
ZERO		SHRU	SUBAB
		SSHL	SUBAH
		SUB	SUBAW
		SUBU	ZERO
		SUB2	
		XOR	
		ZERO	

*：只适用于 S2；**：只适用于 D2。TI 公司许可引用[1,2]。

A.2　浮点操作指令

表 A.2 列出了 C67x 浮点处理器的附加指令，这些指令执行浮点类型的操作。同样，这里也是按照它们所使用的功能单元来进行分类的。

表 A.2　浮点操作指令

.L Unit	.M Unit	.S Unit	.D Unit
ADDDP	MPYDP	ABSDP	ADDAD
ADDSP	MPYI	ABSSP	LDDW
DPINT	MPYID	CMPEQDP	
DPSP	MPYSP	CMPEQSP	
DPTRUNC		CMPGTDP	
INTDP		CMPGTSP	
INTDPU		CMPLTDP	
INTSP		CMPLTSP	
INTSPU		RCPDP	
SPINT		RCPSP	
SPTRUNC		RSQRDP	
SUBDP		RSQRSP	
SUBSP		SPDP	

TI 公司许可引用[1,2]。

参考文献

1. *C6000 CPU and Instruction Set*, SPRU189F, Texas Instruments, Dallas, TX, 2000.
2. *TMS320 TMS320C6000 Programmer's Guide*, SPRU198D, Texas Instruments, Dallas, TX, 2000.

附录 B　循环寻址寄存器和中断寄存器

图 B.1 到图 B.8 介绍了 C6x 处理器中的几个特殊寄存器。

1. 图 B.1 表示的是寻址模式寄存器（AMR），用于循环寻址模式，它用于选择 8 个寄存器指针（A4 至 A7，B4 至 B7）中的一个以及两组用于循环缓冲区的存储区块（BK0，BK1）。
2. 图 B.2 表示的是控制状态寄存器（CSR），其中位 0 是全局中断使能（GIE）位。
3. 图 B.3 表示的是中断启用寄存器（IER）。
4. 图 B.4 表示的是中断标志寄存器（IFR）。
5. 图 B.5 表示的是中断设置寄存器（ISR）。
6. 图 B.6 表示的是中断清除寄存器（ICR）。
7. 图 B.7 表示的是中断服务表指针（ISTP）。
8. 图 B.8 表示的是串行口控制寄存器（SPCR）。

3.7.2 节讨论了 AMR 寄存器；3.14 节讨论了中断寄存器。

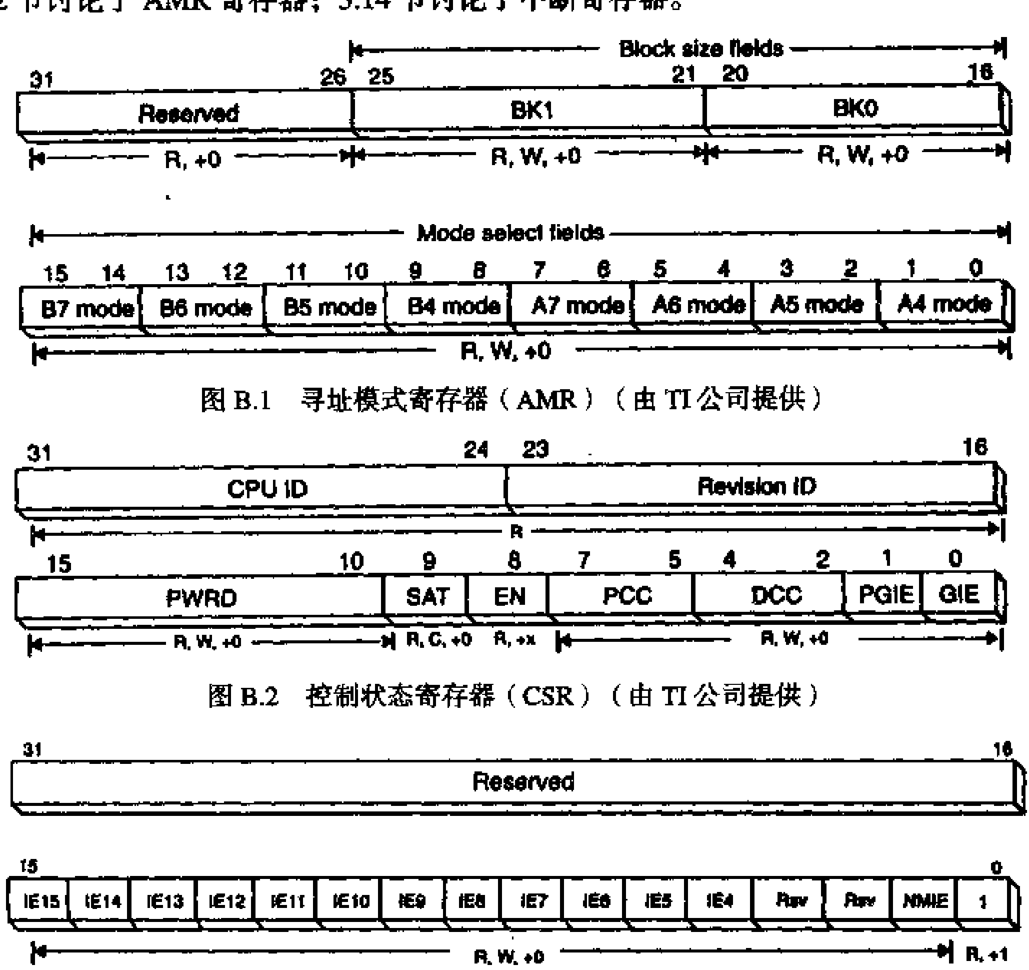


图 B.1　寻址模式寄存器（AMR）（由 TI 公司提供）

图 B.2　控制状态寄存器（CSR）（由 TI 公司提供）

图 B.3　中断启用寄存器（IER）（由 TI 公司提供）

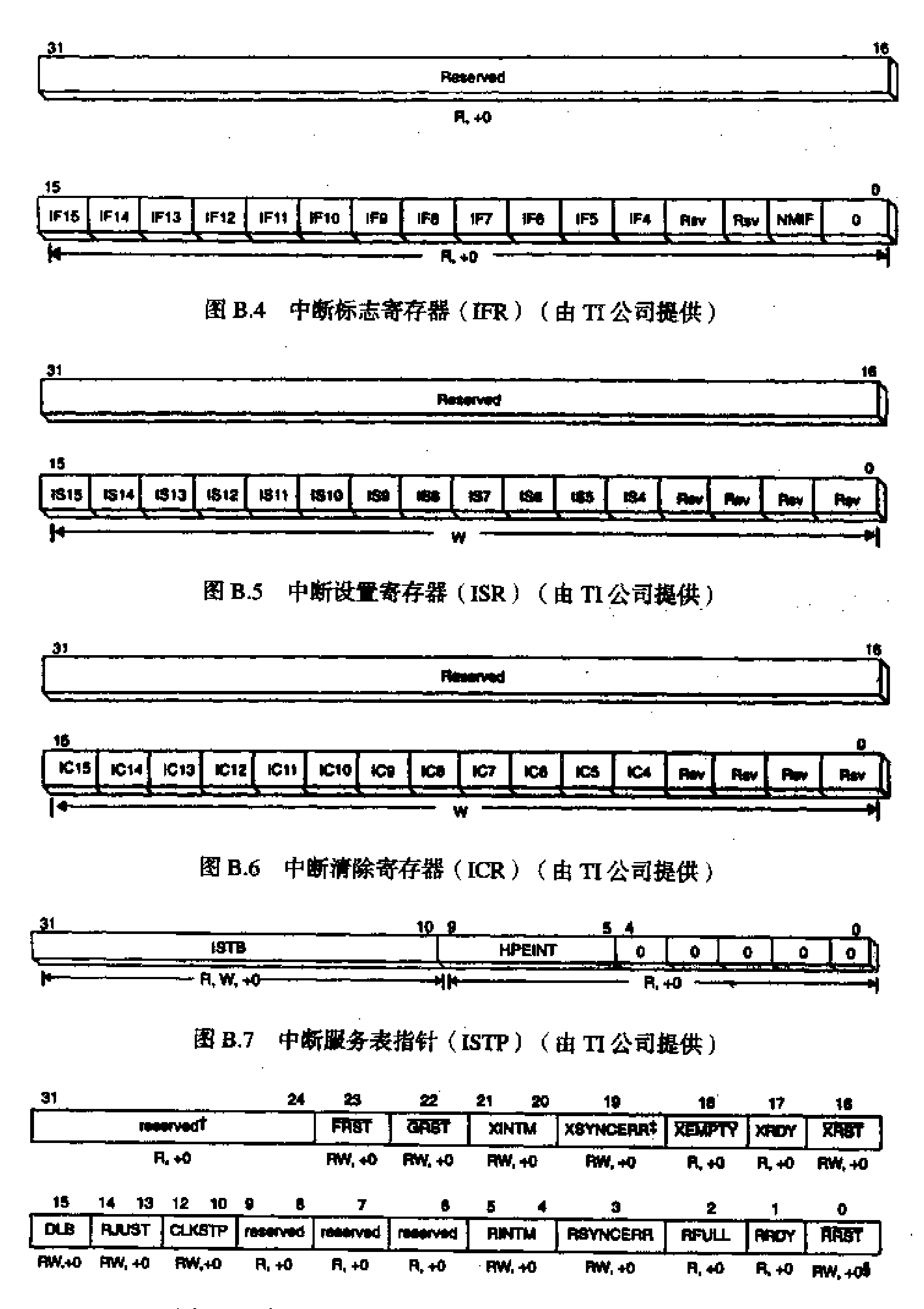


图 B.4　中断标志寄存器（IFR）（由 TI 公司提供）

图 B.5　中断设置寄存器（ISR）（由 TI 公司提供）

图 B.6　中断清除寄存器（ICR）（由 TI 公司提供）

图 B.7　中断服务表指针（ISTP）（由 TI 公司提供）

图 B.8　串行口控制寄存器（SPCR）（由 TI 公司提供）

参考文献

1. *C6000 CPU and Instruction Set*, SPRU189F, Texas Instruments, Dallas, TX, 2000.

附录 C　定点运算需要考虑的问题

C6711 是一款浮点处理器，可以进行整型和浮点运算。C6711 和 AD535 编解码器都支持 2 的补码运算，因此，这里有必要回顾一下定点运算的一些概念[1]。

在定点处理器中，数是用整数表示的；而在浮点处理器中，定点和浮点运算都是可以处理的。和使用定点处理器相比，浮点处理器 C6711 可以表示的数的范围更大。

如果用 2 的补码表示，N 位数的动态范围介于-2^{N-1}至$2^{N-1}-1$之间，也就是说，对于 16 位系统，其表示范围介于–32 768 至 32 767 之间。如果把动态范围归一化到–1 和 1 之间，这样就要把这个范围分成2^N个段，每段的大小是$2^{-(N-1)}$，数据表示的范围就从–1 至$1-2^{-(N-1)}$。对于 4 位系统，则分为 16 段，每段的大小是 1/8，表示的范围从–1 至 7/8。

C.1　二进制和 2 的补码表示

为了更便于说明问题，我们不妨用 4 位字长的系统作为例子，而不是使用 32 位字长。如表 C.1 所示，4 位字可以表示 0 至 15 之间的无符号数。

4 位无符号数可以表示模为 16 的系统，如果把最大数（15）加上 1，该运算将绕回来，从而给出答案 0。有限位系统和号码锁的数字转盘具有同样的模性质，图 C.1 给出了一个数字转盘，数字 0 到 15 围绕在转盘的外面。在该范围内，对任意两个数字 x 和 y，按照如下操作，就可求出两个数字的和：

1. 在转盘上找到第一个数字 x；
2. 沿着顺时针方向前进 y 个单位段，就得到了最终的结果。

表 C.1　无符号二进制数

二进制	十进制
0000	0
0001	1
0010	2
0011	3
.	.
.	.
.	.
1110	14
1111	15

例如：考虑加法 $(5+7) \bmod 16$，得到结果是 12。计算过程是这样的：在转盘上找到 5，然后沿顺时针方向前进 7 个单位段，就得到了结果 12。举另外一个例子：$(12+10) \bmod 16=6$，在数字转盘上找到 12，然后顺时针方向前进 10 单位段，经过 0，得到结果 6。

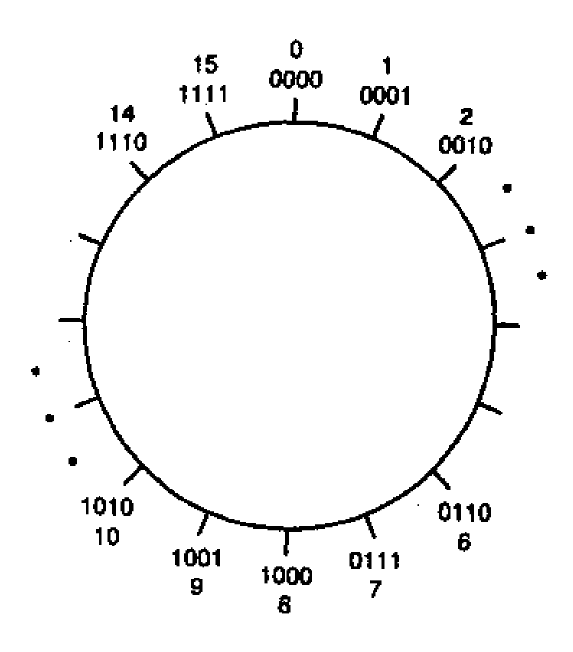


图 C.1　无符号整数的数字转盘

负数在数字转盘上需要对转盘上的数有不同的解释，如果我们穿过数字 8 画一条线并将数字转盘分成两半，右半部分代表正数，左半部分代表负数，如图 C.2 所示，这就表示了一个 2 的补码系统。负数是正数的 2 的补码，反之亦然。

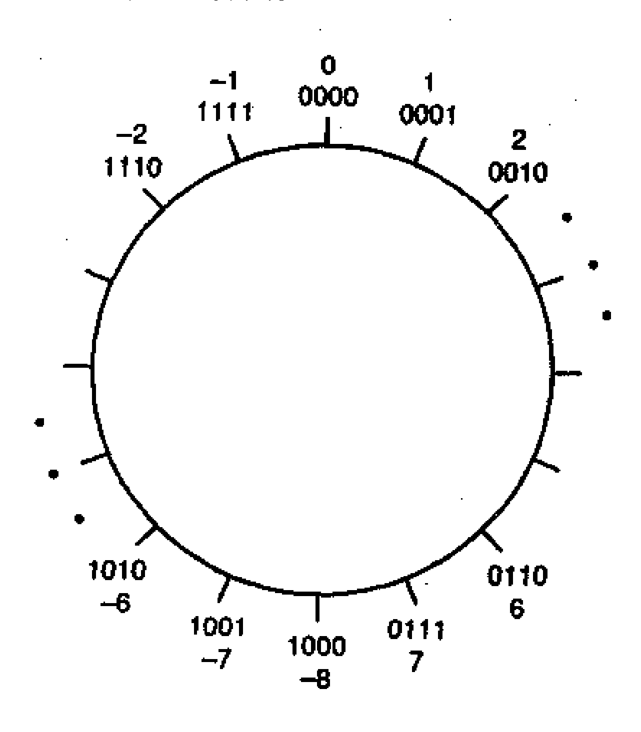


图 C.2　有符号整数的数字转盘

一个二进制整数的 2 的补码表示为：

$$B = b_{n-1} \cdots b_1 b_0$$

它等于十进制整数：

$$I(B) = -b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

其中，$b_{n-1}, \cdots, b_1, b_0$是二进制数。符号位的权值为负，其他所有位的权值都为正。例如：考虑数-2：

$$1110 = -1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -8 + 4 + 2 + 0 = -2$$

使用上面的图形方法来计算 $6 + (-2) \bmod 16 = 4$，在数字转盘上找到 6，然后按顺时针方向转动（1110）个单位段，就得到结果为 4。

同样，两个数的二进制加法如下：

$$\begin{array}{r} 0110 \\ 1110 \\ \hline 10100 \\ \text{C} \end{array}$$

可见，在有限字长寄存器算术运算条件下，最高有效位的进位就被略去，进位对应着绕过数字转盘的零点。对于 n 位的数，假设计算结果在可表示的范围-2^{n-1}到$2^{n-1}-1$之间，如果忽略最高有效位的进位，就会得到两个数相加的正确结果。对于前面 4 位数字转盘的例子，结果应该在-8 至 7 之间。当在 4 位系统中，如果将-7 和-8 进行相加，我们就会得到结果为+1，而不是正确的结果-15，因为-15 超出了表示的范围。当两个符号相同的数相加，结果的符号位和原来的数相反时，出现这种情况时是由于产生了溢出。2 的补码减法等效于被减数的 2 的补码再加上另一个数。

C.2 小数的定点表示

前面刚刚讨论过小数的定点表示，也可以使用定点表示小数，小数的定点表示范围在+0.99…到-1 之间。为了用 n 位表示小数，小数点必须左移 $n-1$ 位，这样就剩下一个符号位和$n-1$个小数位，其表示式：

$$F(B) = -b_0 \times 2^0 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_{n-1} \times 2^{-(n-1)}$$

公式把二进制的小数转换成十进制的小数。和整数的定点表示相似，符号位的权重是-1，其他位的权重是 1/2 的正数次幂。图 C.3 给出了采用数字转盘表示 2 的补码的 4 位小数。小数的数值可用图 C.2 的 2 的补码整数除以2^3得到，因为 4 位系统的位数较少，所以它能表示的范围也比较小，从-1 至 0.875。对 16 位的字，有符号整数的表示范围从-32 768 至+32 767，为了得到小数的表示范围，用 2^{15} 或 32 768 去除这两个有符号数，就可以得出表示范围是从-1 至 0.999 969（通常取 1）。

C.3 乘法

如果将两个 n 位的数相乘，通常的想法是运算结果是 $2n$ 位的操作数，尽管这对无符号数是正确的，但对有符号数就不正确了。如前所述，有符号数需要符号位，符号位的权重是-2^{n-1}，其余的位是正的，权重为 2 的幂指数。为了得到两个 n 位数相乘的结果需要的位数，我们把两个最大的 n 位数相乘：

$$P = (-2^{n-1})(-2^{n-1}) = 2^{2n-2}$$

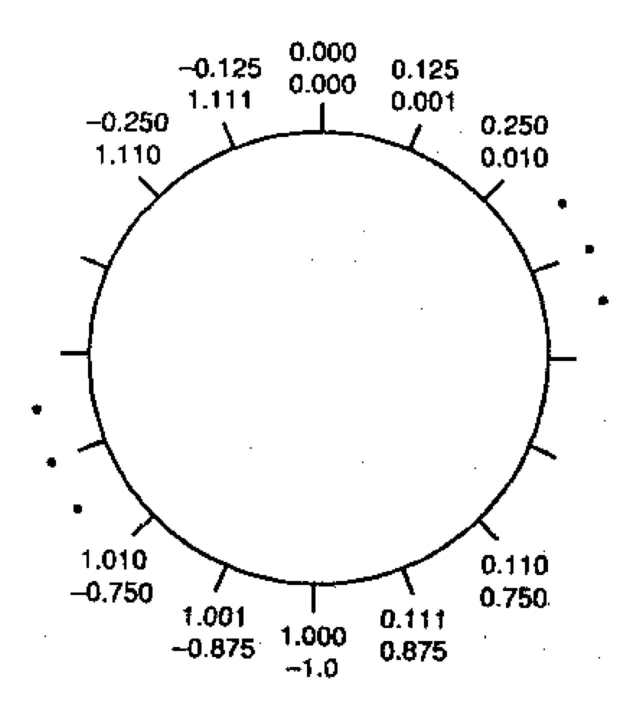


图 C.3　表示定点数的数字转盘

这个数是正数，可以用$2n-1$位表示。如果从 0 比特位开始计算，那么结果的最高位应该是$2n-2$比特位。因为这个数是正数，所以它的符号位不会显示出来；而当它为负数时，符号位就会显示出来（2 的幂指数）。这是一个例外的情况，在用小数表示时，这种情况将被处理为溢出。因为小数表示要求操作数和结果有同样的范围，即大于等于-1 且小于+1。而$(-1)\times(-1)$两个数相乘得到+1，这个数不能用小数来表示。

下面考虑仅次于上面两个数相乘的结果：

$$P=(-2^{n-1})(-2^{n-1}+1)=2^{2n-2}-2^{n-1}$$

式中，因为是第一个数减去第二个数，所以要从 0 位开始计算，结果将会占用$2n-3$个比特位，这样，就可以用$2n-2$ 位表示。除例外情况，第$2n-2$比特位可以作为符号位使用，因此，为了支持$(n\ x\ n)$位有符号数乘法，需要 $2n-1$ 位。

为了阐明前述的等式，考虑下面对于 4 位的情况：

$$P=(-2^3)(-2^3+1)=2^6-2^3$$

数2^6占用第 6 个比特位。因为第二个数是负值，所以两个数的和占据的位只能比比特位 6 少的位，或者如下表示：

$$2^6-2^3=64-8=56=00111000$$

这样，第 6 个比特位置就可以作为符号位，用 8 位等价的数来表示时就有两个符号位（位 6 和位 7）。C6x 支持有符号数和无符号数乘法，因此可用 $2n$ 位来存放乘积。

考虑两个 4 位小数的乘法，每个数有 3 个小数位和一个符号位，结果用一个 8 位数来表示，第一个数是-0.5，第二个数是 0.75，按照下面的过程进行相乘：

$$
\begin{array}{r}
-0.50 = 1.100 \\
\times 0.75 = 0.110 \\
\hline
\underline{11}111000 \\
\underline{1}11000 \\
\hline
111.101000 \\
C \\
= -2^1 + 2^0 + 2^{-1} + 2^{-3} = -0.375
\end{array}
$$

其中，被乘数中加下划线的位表示符号扩展位，当负的被乘数加到部分积上时，必须在左面进行符号位扩展，扩展到乘积的最大比特位。为了说明符号扩展，得出正确的扩展位数，从0开始逆时针方向查看图C.2的数字转盘，写出5位、6位、7位……负数的扩展表示形式。注意，通过对现有4位进行符号扩展得到正确的结果，因此，符号扩展可得到正确的扩展位数。尽管将忽略进位输出，但是，111.101000（9位字），11.101000（8位字）和1.101000（7位字）都表示同样的-0.375，这样，前面例子中的乘积在4位系统中可以用$2n-1$或7位来表示。

当两个16位数相乘产生32位结果时，其实只要31位就可以了，因此，位31是位30的符号扩展位，扩展位通常称为符号位。

考虑下面的例子：把$(0101)_2$和$(1110)_2$相乘，等价于十进制中的5乘以-2，得到的结果是-10，它超出了4位系统的动态范围{-8，7}；当使用3位小数位格式，这相当于0.625乘以-0.25，得到的结果是-0.156 25，结果在小数的表示范围内。

当两个15位小数位格式的数（每个数有一个符号位）相乘时，结果是30位小数位格式的数，有一个多余的符号位，最高位是多余的符号位，可以右移15位来保留最高一些有效位和两个符号位中的一个，通过右移15位（除以2^{15}）可以把结果存储在一个16位的系统中，这样就丢弃了15个最低位，因此会损失一些精度。保留最高15位可以保持高的精度。对于32位的系统，左移一位就可以去掉多余的符号位。

注意到当两个15位小数位格式的数相乘，表示的范围是-1到1，结果保持在同样的范围内；但两个15位小数位的数相加，结果就可能超过此范围，从而引起溢出，因此要使用定标方法纠正这种溢出现象。

因为AD535是16位系统，32位的结果最终必须截短或者四舍五入，最高一些有效位、符号位以及它的扩展位存储在C6x累加寄存器的高位部分，累加寄存器的高位部分左移去掉多余的符号位，当数据传送到16位存储单元时，就增加一个附加分辨率比特位。

参考文献

1. R. Chassaing and D. W. Horning, *Digital Signal Processing with the TMS320C25*, Wiley, New York, 1990.

附录 D　MATLAB 支持工具

附录 D 描述了 MATLAB[1,2]中的几个支持工具：

1. 滤波器设计工具 SPTOOL，它使用图形用户接口（GUI）进行 FIR 和 IIR 滤波器的设计。RTSPTOOL 是 SPTOOL 工具的扩展。
2. 使用学生版 MATLAB 中的函数进行 FIR 和 IIR 滤波器设计。
3. 双线性变换。
4. FFT 和 IFFT。

D.1　用 MATLAB GUI 中的 SPTOOL 设计 FIR 滤波器

MATLAB 提供了图形用户接口（GUI）的滤波器设计工具 SPTOOL，可用于设计 FIR（或 IIR）滤波器。

例 D.1　用 MATLAB GUI 中的滤波器设计工具 SPTOOL 设计 FIR 滤波器

1. 在 MATLAB 中键入：

   ```
   >>sptool
   ```

   进入 MATLAB 图形用户接口滤波器设计工具 SPTOOL，用它设计 FIR 和 IIR 滤波器。
2. 在启动窗口 startup.spt 中，选择一个新的设计并用图 D.1 显示的滤波器特性，设计一个中心频率在 2700 Hz 的 FIR 带阻滤波器。该滤波器有 $N = 89$ 系数（MATLAB 显示的阶数是 $N-1$）。使用凯塞（Kaiser）窗函数，在例 4.1 中，对滤波器的实时实现进行了测试。
3. 测试结束后，再次打开 startup 窗口，选择菜单 Select→Edit→Name，把名字（输入新的文件名）修改为 bs2700。
4. 选择菜单 File→Export→Export to Workspace，把设计的滤波器 bs2700 的设计参数导出到 MATLAB 工作窗口。
5. 进入 MATLAB 工作窗口，键入下面两个命令：

   ```
   >>bs2700.tf.num;
   >>round(bs2700.tf.num*2^15)
   ```

   得到传输函数中分子的系数，并用 2^{15} 进行定标，这时工作窗口显示 FIR 带阻滤波器的系数如下：

   ```
   -14  23  -9···23  -14
   ```

   如图 D.2 所示，这些系数保存在文件 bs2700.cof 中，例 4.1 使用了这些系数。

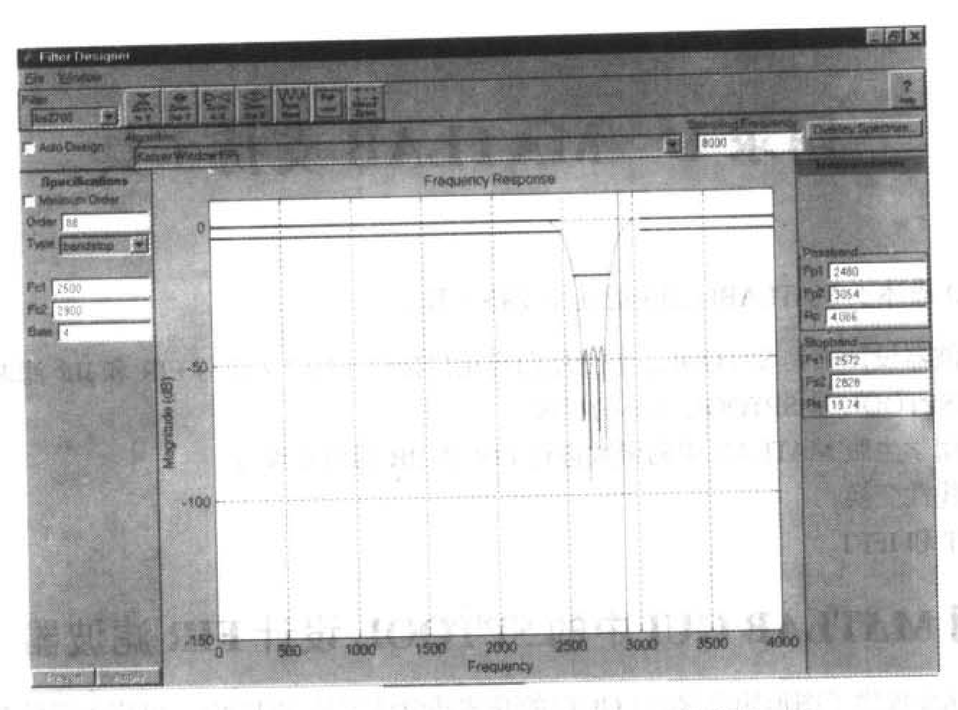


图 D.1 MATLAB 滤波器设计工具 SPTOOL 的窗口，显示中心频率为 2700 Hz 的 FIR 带阻滤波器的特性

```
//BS2700.cof  FIR bandstop coefficients designed with MATLAB

#define N 89                                  //number of coefficients

short h[N]={-14,23,-9,-6,0,8,16,-58,50,44,-147,119,67,-245,200,72,
-312,257,53,-299,239,20,-165,88,0,105,-236,33,490,-740,158,932,-1380,
392,1348,-2070,724,1650,-2690,1104,1776,-3122,1458,1704,29491,1704,
1458,-3122,1776,1104,-2690,1650,724,-2070,1348,392,-1380,932,158,-740,
490,33,-236,105,0,88,-165,20,239,-299,53,257,-312,72,200,-245,67,119,
-147,44,50,-58,16,8,0,-6,-9,23,-14};
```

图 D.2 用 MATLAB 的滤波器设计工具 SPTOOL 设计，中心频率为 2700 Hz 的 FIR 带阻滤波器的系数文件(bs2700.cof)

实时 SPTOOL（RTSPTOOL）

实时 SPTOOL（RTSPTOOL）为 DSK[3-5]提供了一个直接的接口，通过该接口可以在 DSK 上实时地设计和实现（在 MATLAB 环境下）滤波器。RTSPTOOL 的窗口和 SPTOOL 的窗口很相像，只是附加了一个用在 DSK 上实时运行滤波器设计的工具条。按下适当的工具条，就可得到想要的滤波器。系数经过定标后，存到通用 FIR 程序包含的文件中。通过修改 MATLAB 文件 filtdes.m，得到从 MATLAB 到 DSK 的接口。一个 MATLAB 的.m 函数访问 CCS 程序代码生成工具，在 DSK 中编译/汇编、连接、装载/运行可执行文件（装载/运行使用 dsk6xldr filename.out 文件）。

D.2 用 MATLAB GUI 中的 SPTOOL 设计 IIR 滤波器

在 D.1 节中，介绍了利用 MATLAB 的 GUI 滤波器设计工具 SPTOOL 设计 FIR 滤波器的过程，同样的一些过程也可以用于 IIR 滤波器的设计。

例 D.2　用 MATLAB 的 GUI 滤波器设计工具 SPTOOL 设计 IIR 滤波器

图 D.3 显示了用 MATLAB 的滤波器设计工具 SPTOOL 设计出的十阶 IIR 带阻滤波器的特性，其中心频率是 1750 Hz。MATLAB 显示的阶数是 5，表示的是二阶单元的数目。把它保存到文件 bs1750（见例 D.1）中。像前面的 FIR 滤波器设计一样，把滤波器的系数导出到工作窗口，在 MATLAB 的工作窗口键入命令：

```
>>[z,p,k] = tf2zp(bs1750.tf.num, bs1750.tf.den);
>>sec_ord_sec = zp2sos(z,p,k);
>>sec_ord_sec = round(sec_ord_sec*2^15)
```

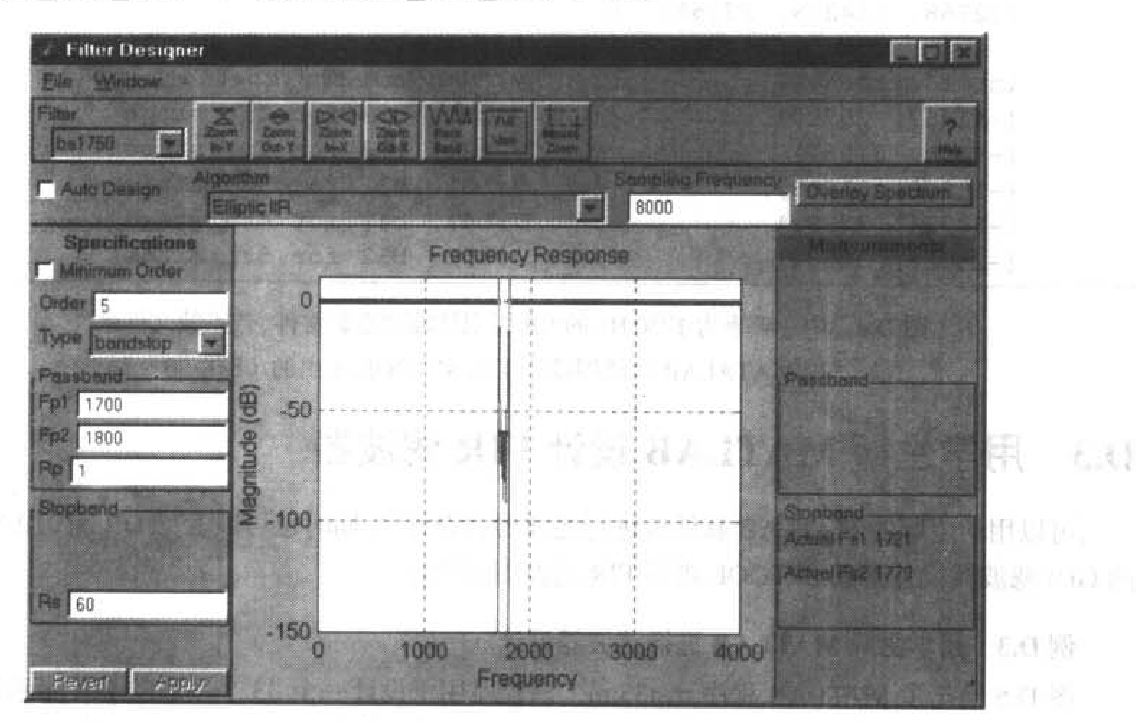


图 D.3　MATLAB 的滤波器设计工具 SPTOOL 窗口显示的中心频率为 1750 Hz 的 IIR 带阻滤波器的特性

第一条命令分别计算出分子和分母的根（零点和极点），并把（定标后的）结果转换成以二阶单元实现的格式。分子分母系数如下：

```
27940    -10910    27940    32768    -11417    25710
.
.
.
32768    -14239    32768    32768    -15258    32584
```

这 30 个系数表示分子的系数 a_0，a_1，a_2 和分母的参数 b_0，b_1，b_2，它们表示每个二阶单元的 6 个系数，b_0 归一化为 1，并且用 $2^{15} = 32\,768$ 定标，这些系数保存在文件 bs1750.cof 中。图 D.4 中也列出了这些系数，例 5.1 也使用了这些系数。图 D.4 显示了 25 个系数（而不是 30 个系数）。因为系数 b_0 总是归一化为 1，程序中并没有使用。如在 FIR 滤波器设计中的一样，按下 RTSPTOOL 中的一个按钮，实时实现该 IIR 带阻滤波器[3,4]。

```
//bs1750.cof IIR bandstop coefficient file, centered at 1,750 Hz

#define stages 5                    //number of 2nd-order stages

int a[stages][3]=        {          //numerator coefficients
{27940, -10910, 27940},             //a10, a11, a12 for 1st stage
{32768, -11841, 32768},             //a20, a21, a22 for 2nd stage
{32768, -13744, 32768},             //a30, a31, a32 for 3rd stage
{32768, -11338, 32768},             //a40, a41, a42 for 4th stage
{32768, -14239, 32768}   };

int b[stages][2]=        {          //*denominator coefficients
{-11417, 25710},                    //b11, b12 for 1st stage
{-9204, 31581},                     //b21, b22 for 2nd stage
{-15860, 31605},                    //b31, b32 for 3rd stage
{-10221, 32581},                    //b41, b42 for 4th stage
{-15258, 32584}          };         //b51, b52 for 5th stage
```

图 D.4 中心频率为 1750 Hz 的 IIR 带阻滤波器系数文件，该系数文件是用 MATALAB 滤波器设计工具 SPTOOL 求出的（bs1750.cof）

D.3 用学生版 MATLAB 设计 FIR 滤波器

可以用学生版的 MATLAB 软件包进行滤波器的设计[2]，同时也可以参见 D.1 节，用 MATLAB 的 GUI 滤波器设计工具 SPTOOL 进行 FIR 滤波器的设计。

例 D.3 用学生版 MATLAB 进行滤波器设计

图 D.5 给出了 MATLAB 程序 mat33.m，该程序用于设计一个 33 个系数的 FIR 带通滤波器。函数 remez 使用基于 Remez 交换算法和 Chebyshev 逼近理论的 Parks-McClellan 算法。设计的滤波器中心频率为 1 kHz，抽样频率为 10 kHz，频率 v 表示归一化频率变量，定义为 $v = f/F_N$，其中，F_N 为奈奎斯特频率。带通滤波器由以下三个频带表示：

1. 第一个频带（阻带）的归一化频率为 0 至 0.1（0 Hz 到 500 Hz），对应的幅度响应为 0。
2. 第二个频带（通带）的归一化频率为 0.15 至 0.25（750 Hz 到 1250 Hz），对应的幅度响应为 1。
3. 第三个频带（阻带）的归一化频率为 0.3 至奈奎斯特频率 1（1500 Hz 至 5000 Hz），对应的幅度响应为 0。

```
%Mat33.m MATLAB program for FIR Bandpass with 33 coefficients Fs=10 kHz

nu= [0 0.1 0.15 0.25 0.3 1];    %normalized frequencies
mag= [0 0 1 1 0 0];             %magnitude at normalized frequencies
c=remez (32,nu,mag);            %invoke remez algorithm for 33 coeff
bp33=c';                        % coeff values transposed
save matpb33.cof bp33 -ascii;   %save in ASCII file with coefficients
[h,w] =freqz (c,1,256);         %frequency response with 256 points
plot(5000*nu,mag,w/pi,abs(h))   %plot ideal magnitude response
```

图 D.5 用于 FIR 滤波器设计的 MATLAB 程序（mat33.m）

在 MATLAB 中运行该程序，检验设计的理想滤波器的幅度响应，图 D.6 是在 MATLAB 中画出的特性图。注意频率 750 Hz 和 1250 Hz 表示通带频率，分别对应归一化频率 0.15 和 0.25，相应的幅度响应为 1。频率 500 Hz 和 1500 Hz 表示阻带频率，分别对应归一化频率 0.1 和 0.3，相应的幅度响应为 0。最后的归一化频率 1 对应奈奎斯特频率 5000 Hz，相应的幅度响应为 0，程序产生 33 个一组的系数，并以 ASCII 格式保存在系数文件 matbp33.cof 中。

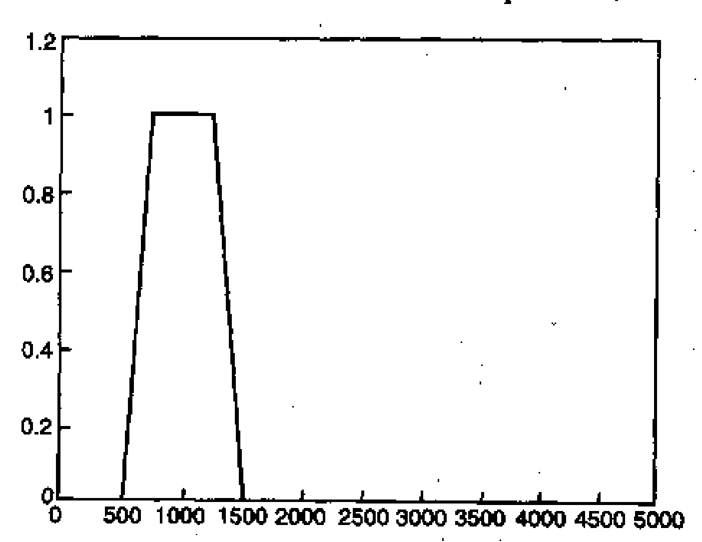


图 D.6 MATLAB 设计的 FIR 带通滤波器的频率响应

例 D.4 用 MATLAB 进行多带 FIR 滤波器的设计

该例将前面的 3 个频带的滤波器例子扩展到有 5 个频带、两个通带的滤波器的设计。程序 mat63.m（见图 D.7）和前面的 MATLAB 程序 mat33.m 类似，滤波器有两个通带，总共由 5 个频带表示：第一个频带（阻带）归一化频率从 0 到 0.1（0 Hz 至 500 Hz），对应的幅度响应是 0；第二个频带（通带）归一化频率从 0.12 到 0.18（600 Hz 至 900 Hz），对应的幅度响应为 1，依次类推。滤波器的参数概述如下：

频带号	频率（Hz）	归一化 f/F_N	幅度响应
1	0~500	0~0.1	0
2	600~900	0.12~0.18	1
3	1000~1500	0.2~0.3	0
4	1600~1900	0.32~0.38	1
5	2000~5000	0.4~1	0

```
%Mat63.m MATLAB program for two passbands, 63 coefficients Fs=10 kHz

nu= [0 0.1 0.12 0.18 0.2 0.3 0.32 0.38 0.4 1]; %normalized frequencies
mag= [0 0 1 1 0 0 1 1 0 0];     %magnitude at normalized frequencies
c=remez (62,nu,mg);              %invoke remez algorithm for 63 coeff
bp63=c';                         % coeff values transposed
save mat2bp.cof bp63 -ascii;     %save in ASCII file with coefficients
[h,w] =freqz (c,1,256);          %frequency response with 256 points
plot (500*nu,mag,w/pi,abs(h))    %plot ideal magnitude response
```

图 D.7 用于两个通带的 FIR 滤波器设计的 MATLAB 程序（mat63.m）

在 MATLAB 中运行该程序，检验图 D.8 所示的两个通带的滤波器的幅度响应（理想状态）。该程序产生 63 个一组的系数，并以 ASCII 格式保存在系数文件 mat2bp.cof 中。

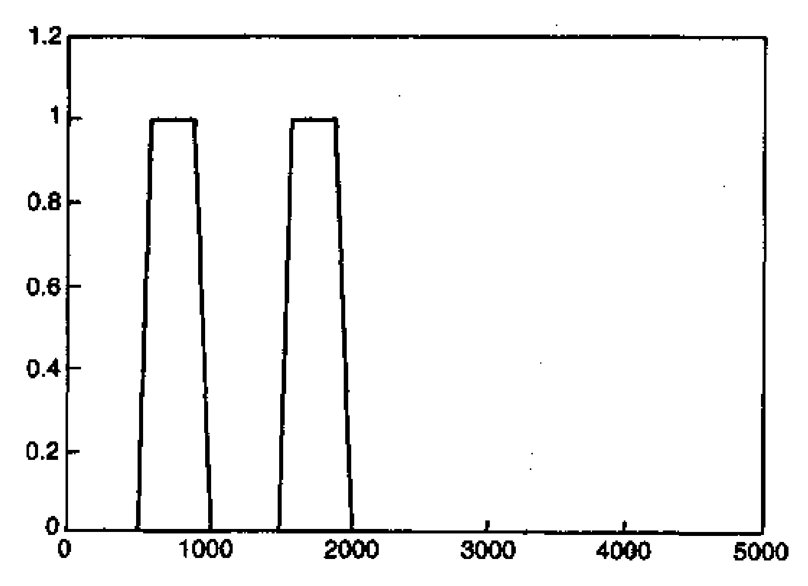


图 D.8 用 MATLAB 设计的两个通带的 FIR 滤波器的频率响应

D.4 用学生版 MATLAB 进行 IIR 滤波器的设计

学生版 MATLAB 也可用于 IIR 滤波器的设计。也可参见 D.2 节，用 MATLAB 的 GUI 滤波器设计工具 SPTOOL 进行 IIR 滤波器的设计。

例 D.5 用学生版 MATLAB 进行 IIR 滤波器的设计

MATLAB 函数 yulewalk 基于最优最小平方拟合方法，可以用于设计递归式滤波器。重新考虑图 D.5 中的 MATLAB 程序 mat33.m，得到一组 33 个系数的 FIR 带通滤波器，其中心频率为 1000 Hz，用下面命令代替 remez 函数进行 FIR 滤波器的设计：

```
>>[a,b] = yulewalk(n,nu,mag)
```

执行这条命令，返回系数 a 和 b 的值，这些系数对应第 5 章讨论的 IIR 滤波器通用输入-输出等式中的系数，滤波器阶数 n 表示二阶单元的个数。例 5.1 的 C 程序利用级联二阶单元实现 IIR 滤波器，这也是最常用的实现方法。例如：如果 yulewalk 函数中 $n = 6$，根据 MATLAB 得到的系数 a 和 b，第 5 章中通用传输函数将三个级联单元变为一个二阶单元。

D.5 用 MATLB 和支持程序及双线性变换方法实现滤波器设计

本节对 5.3 节讨论的双线性变换方法进行了扩展。

练习 D.1 一阶 IIR 低通滤波器

假设一个一阶模拟低通系统传输函数 $H(s)$，可以得到对应的离散时间滤波器的传输函数为 $H(z)$，令带宽或截止频率 $B = 1$ rad/s，抽样频率为 $F_s = 10$ Hz。

1. 选择一个合适的传输函数：

$$H(s)=\frac{1}{s+1}$$

它表示一个带宽为 1 rad/s 的低通滤波器。

2．用下式对 ω_D 进行非线性变换：

$$\omega_A=\tan\frac{\omega_D T}{2}=\tan\left(\frac{1}{20}\right)\approx\frac{1}{20}$$

其中 $\omega_D=B=1$ rad/s，$T=1/10$。

3．消除 $H(s)$ 分子分母公因子，可得：

$$H(s/\omega_A)=\frac{1}{20s+1}$$

4．求得要求的传输函数 $H(z)$，也就是：

$$H(z)=H(s/\omega_A)|_{s=(z-1)/(z+1)}=\frac{z+1}{21z-19}$$

练习 D.2　一阶 IIR 高通滤波器

假设一个高通传输函数 $H(s)=s/(s+1)$，得到相应的传输函数 $H(z)$。令带宽或截止频率为 1 rad/s，抽样频率为 5 Hz，由前面的所述过程，可得 $H(z)$ 为：

$$H(z)=\frac{10(z-1)}{11z-9}$$

练习 D.3　二阶 IIR 带阻滤波器

假设一个二阶带阻滤波器的模拟传输函数为 $H(s)$，可得对应的离散传输函数为 $H(z)$，令低截止频率和高截止频率分别为 950 Hz 和 1050 Hz，抽样频率 F_s 为 5 kHz。

选择的带阻滤波器的传输函数为：

$$H(s)=\frac{s^2+\omega_r^2}{s^2+sB+\omega_r^2}$$

其中 B 和 ω_r 分别为带宽和中心频率，模拟频率为：

$$\omega_{A1}=\tan\frac{\omega_{D1}T}{2}=\tan\frac{2\pi\times 950}{2\times 5000}=0.6796$$

$$\omega_{A2}=\tan\frac{\omega_{D2}T}{2}=\tan\frac{2\pi\times 1050}{2\times 5000}=0.7756$$

带宽 $B=\omega_{A2}-\omega_{A1}=0.096$，$\omega_r^2=(\omega_{A1})(\omega_{A2})=0.5271$，传输函数 $H(s)$ 变为：

$$H(s)=\frac{s^2+0.5271}{s^2+0.096s+0.5271}\tag{D.1}$$

对应的传输函数 $H(z)$ 可由 $s=(z-1)/(z+1)$ 得到，即：

$$H(z)=\frac{[(z-1)/(z+1)]^2+0.5271}{[(z-1)/(z+1)]^2+0.096(z-1)/(z+1)+0.5271}$$

也可以简化为：

$$H(z)=\frac{0.9408-0.5827z^{-1}+0.9408z^{-2}}{1-0.5827z^{-1}+0.8817z^{-2}} \tag{D.2}$$

如后面所示，$H(z)$ 可用程序 BLT.BAS（在辅助材料中）或 MATLAB 进行检验，或者和我们所阐述的一样，用 BLT 方法由 $H(s)$ 计算出 $H(z)$。利用该设计过程进行高阶滤波器的设计也是非常有用的。

练习 D.4 四阶 IIR 带通滤波器

利用 BLT 方法可以设计四阶 IIR 带通滤波器，令低截止频率和高截止频率分别为 1 kHz 和 1.5 kHz，抽样频率为 10 kHz。

1. 四阶巴特沃斯带通滤波器的传输函数 $H(s)$ 可以从二阶巴特沃斯低通滤波器的传输函数得到，即：

$$H(s)=H_{LP}(s)\Big|_{s=(s^2+\omega_r^2)/sB}$$

其中，$H_{LP}(s)$ 是二阶巴特沃斯低通滤波器的传输函数。这样，$H(s)$ 就变为：

$$\begin{aligned}H(s)&=\frac{1}{s^2+\sqrt{2}s+1}\Big|_{s=(s^2+\omega_r^2)/sB}\\&=\frac{s^2B^2}{s^4+\sqrt{2}Bs^3+(2\omega_r^2+B^2)s^2+\sqrt{2}B\omega_r^2s+\omega_r^4}\end{aligned} \tag{D.3}$$

2. 模拟频率 ω_{A1} 和 ω_{A2} 为：

$$\omega_{A1}=\tan\frac{\omega_{D1}T}{2}=\tan\frac{2\pi\times1050}{2\times10{,}000}=0.3249$$

$$\omega_{A2}=\tan\frac{\omega_{D2}T}{2}=\tan\frac{2\pi\times1500}{2\times10{,}000}=0.5095$$

3. 现在可求得中心频率 ω_r 和带宽 B：

$$\omega_r^2=(\omega_{A1})(\omega_{A2})=0.1655$$

$$B=\omega_{A2}-\omega_{A1}=0.1846$$

4. 模拟传输函数 $H(s)$ 可以简化为：

$$H(s)=\frac{0.03407s^2}{s^4+0.26106s^3+0.36517s^2+0.04322s+0.0274} \tag{D.4}$$

5. 相应的 $H(z)$ 变成：

$$H(z)=\frac{0.02008-0.04016z^{-2}+0.02008z^{-4}}{1-2.5495z^{-1}+3.2021z^{-2}-2.0359z^{-3}+0.64137z^{-4}} \tag{D.5}$$

该式是用式（5.4）的形式来表示的，可用 BLT.BAS 程序对其进行检验。

练习 D.5 用 MATLAB 中的双线性函数把 $H(s)$ 转换成 $H(z)$

用练习 D.3 中的二阶 IIR 带阻滤波器，模拟 s 平面的传输函数[见式（D.1）]：

$$H(s)=\frac{s^2+0.5271}{s^2+0.096s+0.5271}$$

可以用 MATLAB 中的双线性函数以及下面的命令将它转换为等价的数字z平面的传输函数，MATLAB 命令如下：

```
>>num = [1, 0, 0.5271];                  %numerator coefficients
>>den = [1, 0.096, 0.5271];              %denominator coefficients
>>T = 2; Fs = 1/T;                       %K=1 from bilinear equation
>>[a,b]=bilinear (num, den, Fs)  %invoke bilinear function
```

得到式（5.4）中的传输函数相应的系数 a 和 b，即：

$$H(z)=\frac{0.9409-0.5827z^{-1}+0.9409z^{-2}}{1-0.5827z^{-1}+0.8817z^{-2}}$$

该传输函数和练习 D.3 所得到的传输函数（D.2）是一样的。注意，为了方便起见，在 MATLAB 中选择 $T=2$，因为在第 5 章中的双线性等式的常数 $K=2/T$ 设为 1。同时注意 MATLAB 中使用的通用输入输出方程：

$$y(n)=b_0x(n)+b_1x(n-1)+b_2x(n-2)+\cdots-a_1y(n-1)-a_2y(n-2)-\cdots$$

由上式可得如下所示的传输函数：

$$H(z)=\frac{b_0+b_1z^{-1}+b_2z^{-2}+\cdots}{1+a_1z^{-1}+a_2z^{-2}+\cdots}$$

上式表明 MATLAB 的系数 a 和 b 与式（5.1）中系数的顺序是相反的。

练习 D.6　用实用程序 BLT.BAS 把 $H(s)$ 转换成 $H(z)$

BLT.BAS 是一个非常有用的程序，它是用 BASIC 编写的，可以把模拟的传输函数 $H(s)$ 用双线性变换公式 $s=(z-1)/(z+1)$ 转换成相应的传递函数 $H(z)$。为了检验练习 D.3 中二阶带阻滤波器式（D.1）的结果，运行 GWBASIC，然后调入并运行程序 BLT.BAS，屏幕上出现如图 D.9 (a) 所示的提示符和与 $H(s)$ 相关的系数 a 和 b，与传输函数 $H(z)$ 相应的系数 a 和 b 如图 D.9(b)所示，这些数据验证了式（D.1）的正确性。再运行程序 BLT.BAS，使用式（D.4）中的数据验证式（D.5）的正确性。

```
Enter the # of numerator coefficients (30 = Max, 0 = Exit) --> 3
     Enter a(0)s^2  --> 1
     Enter a(1)s^1  --> 0
     Enter a(2)s^0  --> 0.5271

Enter the # of denominator coefficients --> 3
     Enter b(0)s^2  --> 1
     Enter b(1)s^1  --> 0.096
     Enter b(2)s^0  --> 0.5271

Are the above coefficients correct ? (y/n) y
```

(a) s 平面的系数

```
a(0)z^-0 =  0.94085      b(0)z^-0 =  1.00000
a(1)z^-1 = -0.58271      b(1)z^-1 = -0.58271
a(2)z^-2 =  0.94085      b(2)z^-2 =  0.88171
```

(b) z 平面的系数

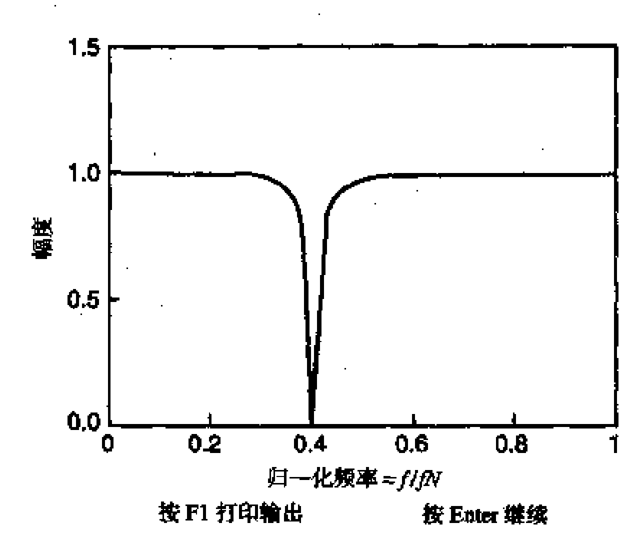
图 D.9　应用程序 BLT.BAS 进行双线性变换

练习 D.7　用实用程序 AMPLIT.CPP 求出幅度和相位响应

使用 C++语言程序 AMPLIT.CPP(在辅助材料中),可以画出最大阶数为 10,传输函数为 $H(z)$ 的给定滤波器的幅度和相位响应。编译（用 Borland C++编译器）并运行该程序，输入练习 D.3 中如图 D.10(a)所示的对应二阶带阻滤波器（D.2）传输函数的系数，图 D.10(b)和图 D.10(c)分别给出了该二阶带阻滤波器的幅度和相位响应。从 $H(z)$ 的幅度响应图中，可以看出归一化中心频率 $\nu = f/F_N = 1000/2500 \ = 0.4$。

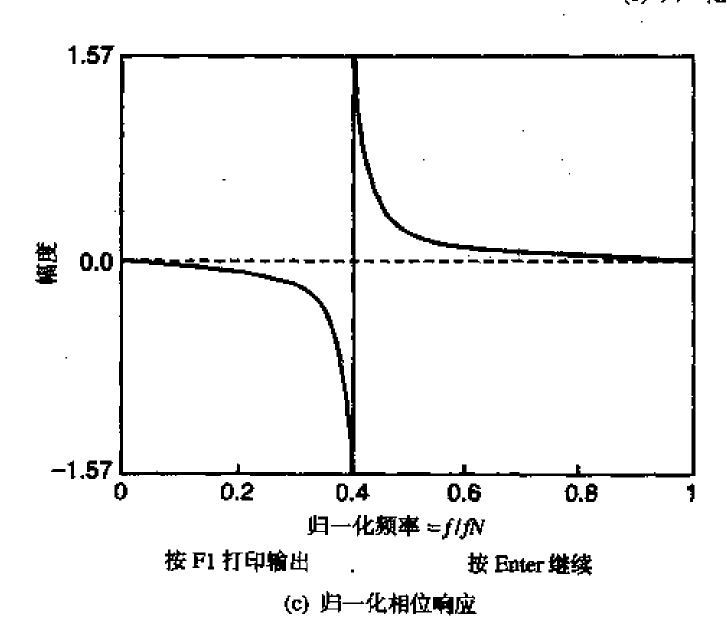
滤波器系数			
分子		分母	
z-0	.9408	z-0	1
z-1	-.5827	z-3	-.5827
z-3	.9408	z-4	.8817
z-4		z-5	
z-5		z-6	
z-6		z-7	
z-7		z-8	
z-8		z-9	
z-9		z-10	
z-10			
F1 HELP	F5 QUIT	F10 PLOT	

(a) z平面的系数





(b) 归一化幅度响应



(c) 归一化相位响应

图 D.10　应用程序 AMPLIT.CPP 画出幅频和相频响应

重新运行该程序，画出练习 D.4 中的四阶带通 IIR 滤波器的幅度响应，检验图 D.11 的结果，归一化中心频率 $\nu = 1250/5000 \approx 0.25$。

应用程序 MAGPHSE.BAS（在辅助材料中）是用 BASIC 语言编写的，可用该程序把幅度和相位响应制成表格的形式。

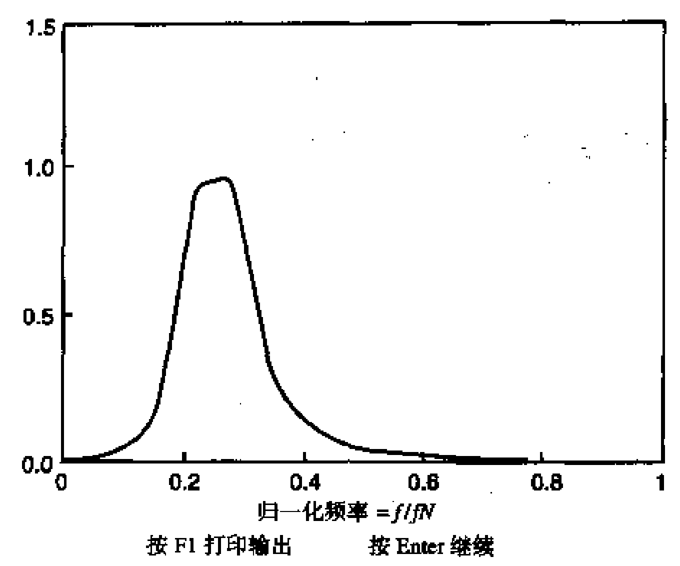


图 D.11 程序 AMPLIT.CPP 画出的四阶带通 IIR 滤波器的幅度响应

D.6 FFT 和 IFFT

MATLAB 可用来求序列的快速 Fourier 变换以及 Fourier 逆变换 IFFT。

练习 D.8 用 MATLAB 实现 8 点的 FFT 和 IFFT

练习 6.1 中的 8 点 FFT 可以很容易地用 MATLAB 进行检验，使用下面的命令：

```
>>x = [1 1 1 1 0 0 0 0];
>>y = fft(x)
>>magy = abs(y)
>>plot (magy)
```

FFT 变换的幅度谱也用图形的方式给出了。

同样地，也可以检验 IFFT，假定练习 6.1 为输出序列，用 IFFT 可以求得：

```
>>X = [4 1-2.414*i 0 1-0.414*i 0 1+0.414*i 0 1+2.414*i];
>>y = ifft(X)
```

这里，矩形序列 y 就是求得的结果。

参考文献

1. *MATLAB, The Language of Technical Computing*, MathWorks, Natick, MA 2000.
2. *MATLAB Student Version*, MathWorks, Natick, MA.
3. W. J. Gomes III and R. Chassaing, Filter design and implementation using the TMS320C6x interfaced with MATLAB, *Proceedings of the 1999 ASEE Annual Conference*, 1999.
4. W. J. Gomes III and R. Chassaing, Real-time FIR and IIR filter design using MATLAB interfaced with the TMS320C31 DSK, *Proceedings of the 1999 ASEE Annual Conference*, 1999.
5. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, Wiley, New York, 1999.

附录 E　其他的支持工具

还有下面的一些支持工具可以利用（参见附录 D 中的 MATLAB 支持）:

1．用于产生信号的 Goldwave 工具软件、虚拟仪器等。
2．MultiDSP 公司用于 FIR 和 IIR 滤波器设计的 DigiFilter 工具。
3．自制的滤波器开发包。
4．可视化应用程序开发环境软件（VAB）。
5．集成 DSP 的编解码器支持工具。
6．MATLAB 的开发工具包。

E.1　作为虚拟仪器的 Goldwave 共享工具

Goldwave 是一个共享工具软件，它可以把 PC 声卡变成一个虚拟仪器。Goldwave 可以从 Web 上下载[1]，该工具通过生成一个函数发生器来产生不同的信号，比如正弦信号和随机噪声，也可以用做示波器或频谱分析仪，并且能记录、编辑一段语音信号，可以得到如回声和滤波等声音效果。低通、高通、带通和带阻滤波器都可以用 Goldwave 工具在声卡上实现，也可以很方便地在信号上加上各种效果。

Goldwave 可用来得到一段语音（TheForce.wav，见辅助材料），加上频率分别为 900 Hz 和 2700 Hz 的两个正弦信号，得到一个受干扰的语音信号，如图 4.24 所示。例 4.7 说明了如何消除这两个正弦信号。

在 Windows 9x 上可以运行两个 Goldwave 程序，一个用来产生 DSK 的输人信号，另一个作为 DSK 输出到声卡的频谱分析仪，但是运行两个 Goldwave 程序会使得到的结果有很多噪声。

其他的共享程序，如 Cool Edit[2]或 Spectrogram[3]，也可以用做虚拟频谱分析仪。

E.2　用 DigiFilter 进行滤波器设计

DigiFilter 是一个滤波器设计包，可用于 FIR 和 IIR 滤波器设计[4]。现在，它可以和 C31 的 DSK 接口，实时实现滤波器的设计。

E.2.1　FIR 滤波器设计

图 E.1 显示的是含有 61 个系数的 FIR 带通滤波器的对数幅度响应，其中心频率为 2 kHz，用凯塞窗函数实现。对于特定的设计，根据每种窗函数（矩形窗、汉明窗等）的阶数（系数）指标，用户可以从几种窗函数中选择合适的窗函数。如图 E.2 所示，脉冲响应和阶跃响应都可以得到。注意用汉明窗函数实现需要 89 个系数，然而用凯塞窗函数实现只需要 61 个系数（见图 E.2）。

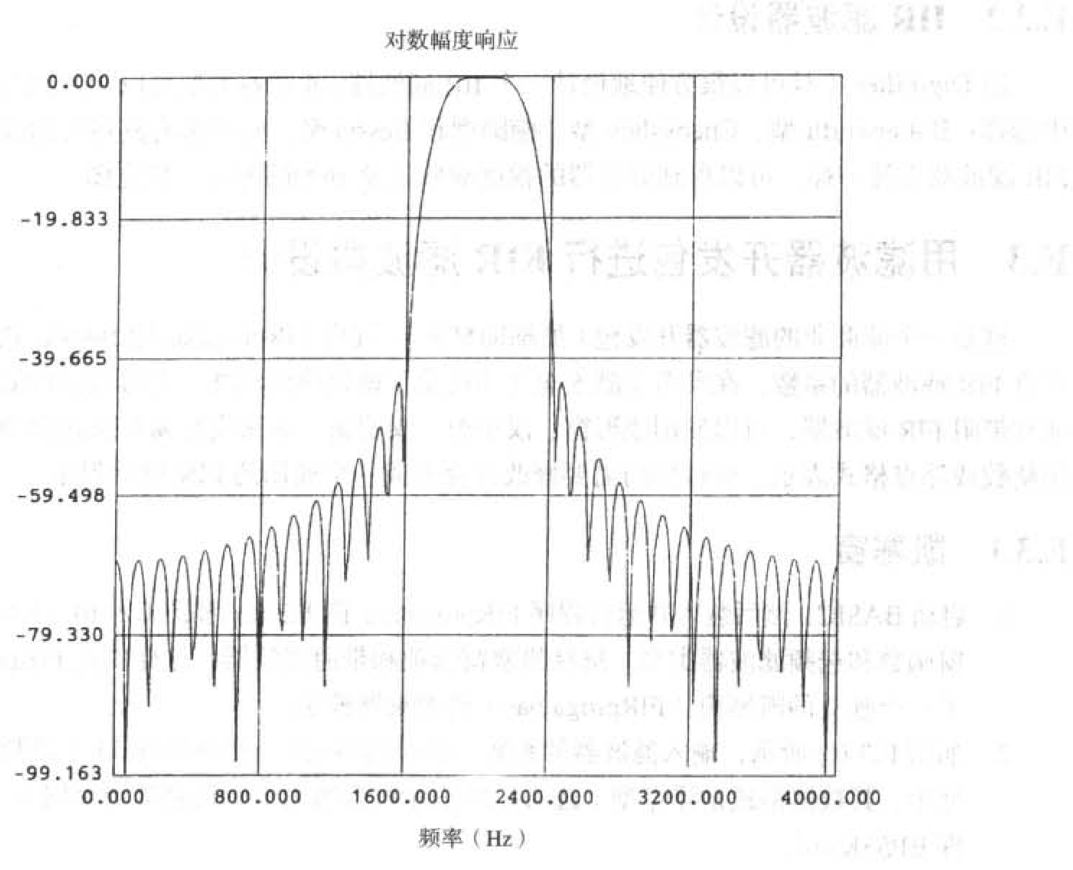


图 E.1 用 DigiFilter 设计的 FIR 带通滤波器的幅度响应

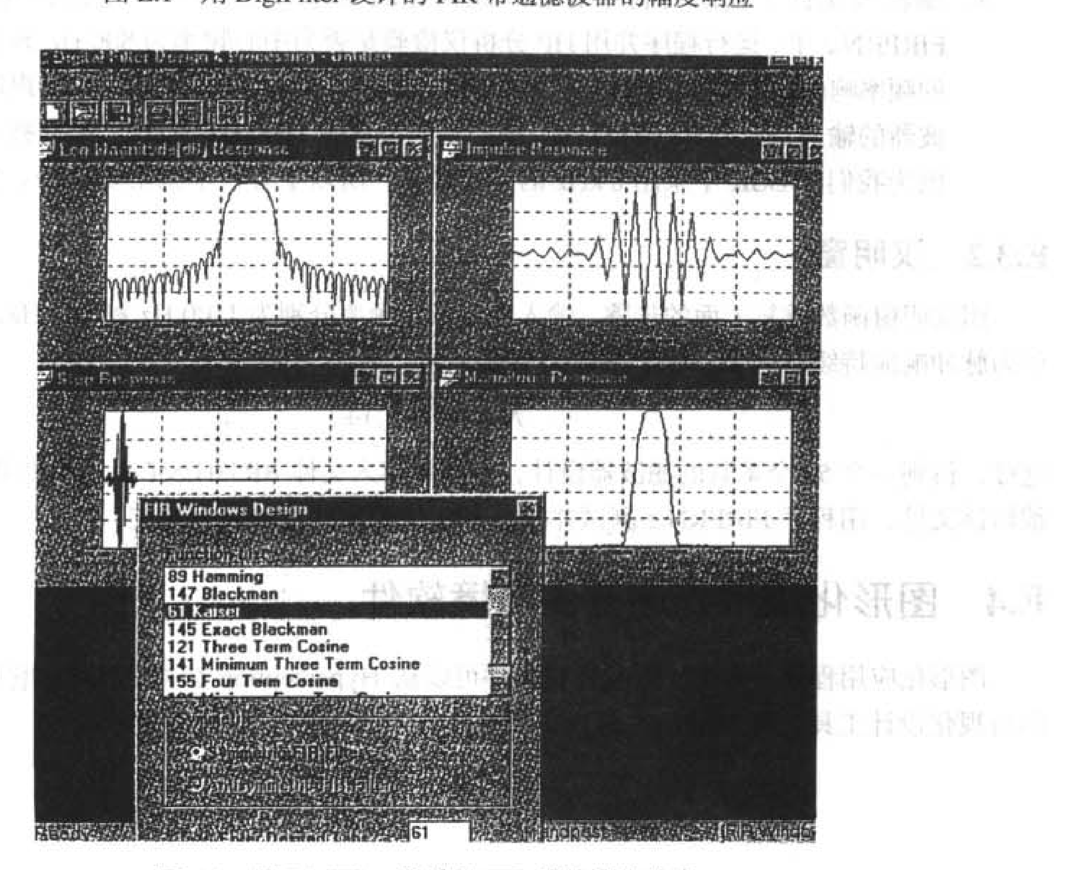


图 E.2 用 DigiFilter 设计的 FIR 滤波器的响应

E.2.2　IIR 滤波器设计

用 DigiFilter 工具可以很方便地设计一个 IIR 滤波器。滤波器类型可以从下面几种滤波器类型中选择：Butterworth 型、Chebyshev 型、椭圆型和 Bessel 型，每种都有其特定的滤波器阶数。和 FIR 滤波器设计一样，可以得到滤波器的幅度响应以及 $H(z)$的零点、极点图。

E.3　用滤波器开发包进行 FIR 滤波器设计

这是一个非商业的滤波器开发包（见辅助材料），程序 FIRprog.bas 用 BASIC 语言写成，可以计算 FIR 滤波器的系数，在参考文献 5 至 7 中讨论了该程序的功能。它可以设计低通、高通、带通和带阻 FIR 滤波器，可以使用矩形窗、汉宁窗、汉明窗、布莱克曼窗和凯塞窗函数，结果可以用整数或浮点格式表示。该程序尚需要修改并合并成一个通用的 FIR 设计程序。

E.3.1　凯塞窗

1. 启动 BASIC，然后装入并运行程序 FIRprog.bas，图 E.3 (a) 和图 E.3 (b) 显示了可以选择的窗函数和选频滤波器类型。选择凯塞窗选项和带通滤波器，这样就在 FIRprog.bas 中调用了一个独立的凯塞窗（FIRproga.bas）的滤波器模块。
2. 如图 E.3 (c) 所示，输入滤波器的系数，选择选项 c31，把所得的 53 个系数保存到一个文件中，其数据格式是浮点型（选项 C25 表示把系数用十六进制进行存储）。把结果存入文件 BP53K.cof。
3. 编辑该文件（辅助材料中有已经编辑好的文件），把该程序包含到例 4.4 中的程序 FIRPRN.c 中，运行程序并用 HP 分析仪检验是否为中心频率为 800 Hz 的 FIR 带通滤波器的频率响应，结果如图 E.4。在程序 FIRPRN.c 中，用内部产生的一个噪声序列作为 FIR 滤波器的输入，在该滤波器设计时，中心频率设置为 1000 Hz(F_s/10)，抽样频率为 10 000 Hz。因为我们在 DSK 中采用 8 kHz 的抽样频率，所以中心频率就是 800 Hz，如图 E.4 所示。

E.3.2　汉明窗

用汉明窗函数重复上面的步骤，输入高低截止频率分别为 1100 Hz 和 900 Hz，输入 5.2（ms）作为脉冲响应持续时间 D，因为系数的个数 N 为

$$N=(D\times F_s)+1$$

这样，得到一个 53 个系数的滤波器设计，把结果存入文件 BP53H.cof 中。和使用凯塞窗时一样，编辑该文件，用程序 FIRPRN.c 测试并检验 FIR 带通滤波器的频率响应。

E.4　图形化应用程序开发环境软件

图形化应用程序（VAB）开发环境软件可以从 Hyperception 公司[8]买到，它是一个基于组件的可视化设计工具，可以用来实现 DSP 算法。

```
                                Main Menu
                                ---------

                                1. . . .RECTANGULAR
                                2. . . .HANNING
                                3. . . .HAMMING
                                4. . . .BLACKMAN
                                5. . . .KAISER
                                6. . . .Exit to DOS

          Enter window desired (number only) -> 5
```

(a)

```
                    Selections:
                              1. . . .LOWPASS
                              2. . . .HIGHPASS
                              3. . . .BANDPASS
                              4. . . .BANDSTOP
                              5. . . .Exit back to Main Menu

        Enter desired filter type (number only) -> 3
```

(b)

```
     Specifications:
                  BANDPASS
                  Passband Ripple (AP) = 6 db
                  Stopband Attenuation (AS) = 30 db
                  Lower Passband Frequency = 900 Hz
                  Upper Passband Frequency = 1100 Hz
                  Lower Stopband Frequency = 600 Hz
                  Upper Stopband Frequency = 1400 Hz
                  Sampling Frequency (Fs) = 10000 Hz

     The calculated # of coefficients required is: 53

     Enter # of coefficients desired ONLY if greater than 53
     otherwise, press <Enter> to continue ->
```

(c)

```
Send coefficients to:
                  (S)creen
                  (P)rinter
                  (F)ile: contains TMS320 (C25 or C31) data format
                  (R)eturn to Filter Type Menu
                  (E)xit to DOS

Enter desired path --> f

Enter DSP type (C25 OR C31):? c31
```

(d)

图 E.3 用滤波器设计软件包设计 FIR 滤波器：(a) 选择窗函数；(b) 滤波器类型；(c) 滤波器特性；(d) 系数格式菜单

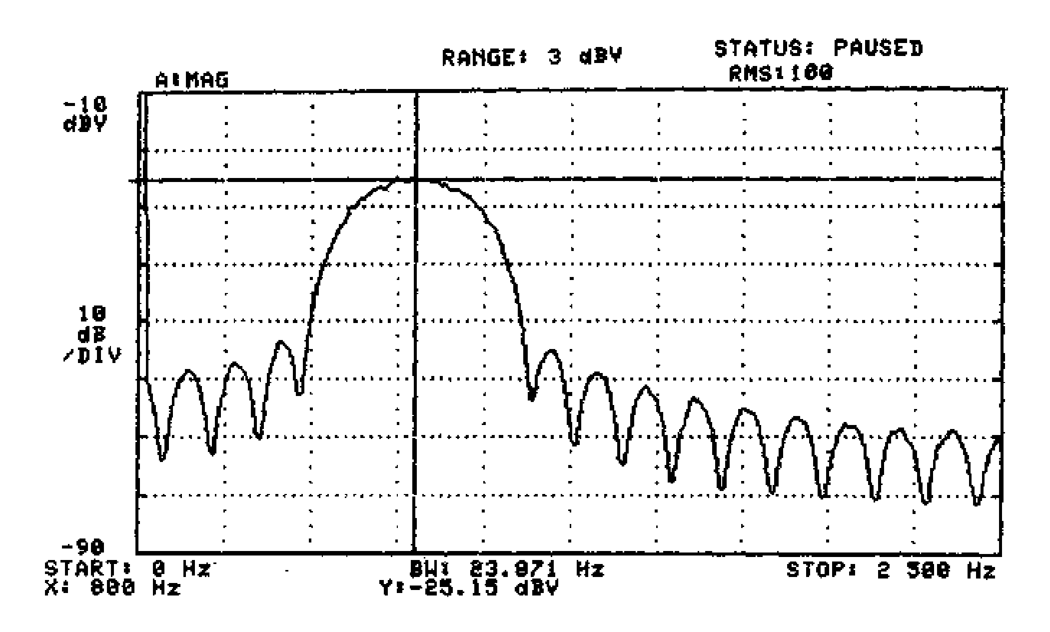


图 E.4 系数文件 BP53K.cof 对应的带通 FIR 滤波器的频率响应

VAB 的使用方法很简单，可以通过用鼠标连接不同的组件来方便地实现 DSP 算法。用户只需要选择相应的函数，把这些函数放进工作表中，选择它们相互间的参数，并且用连线来描述数据流即可。设计方法非常像画一张系统的设计框图，基于 DSP 的设计和实现可以在 DSP 硬件上直接产生和运行，不需要写任何代码。

VAB 包括使用范围很宽的函数模块组件，可以完成 FFT、滤波等，并且支持 C6711 DSK。在几分种内就可以设计和测试一个 DSP 系统，包括的函数模块有信号发生器、A/D 和 D/A、滤波器、FFT、图形处理组件等。设计的结果可以很快地在 PC 的显示器上显示出来，就如象算法是在外部运行而在示波器上显示一样。图 E.5 显示了一个 Vocoder 在 C6711 DSK 上的实现框图。

E.5 其他小的支持工具

我们可以得到下面的支持工具（也可参见附录 D 的 MATLAB 和附录 F 的基于 PCM3003 编解码器的音频子板）：

1. 基于 AD77 立体声编解码器的音频子板，可以接口到 C6x 的 DSK 上，可以从集成 DSP 公司买到[9]。
2. TI 公司的 DSP 开发工具包[10]，可以通过 TI 公司的软件和硬件把 MATLAB 和 SIMULINK 连接起来，它注重代码的优化、测试和分析，而不是重写 DSP 算法。现在它支持基于 C6701 的评估版模块（EVM）。

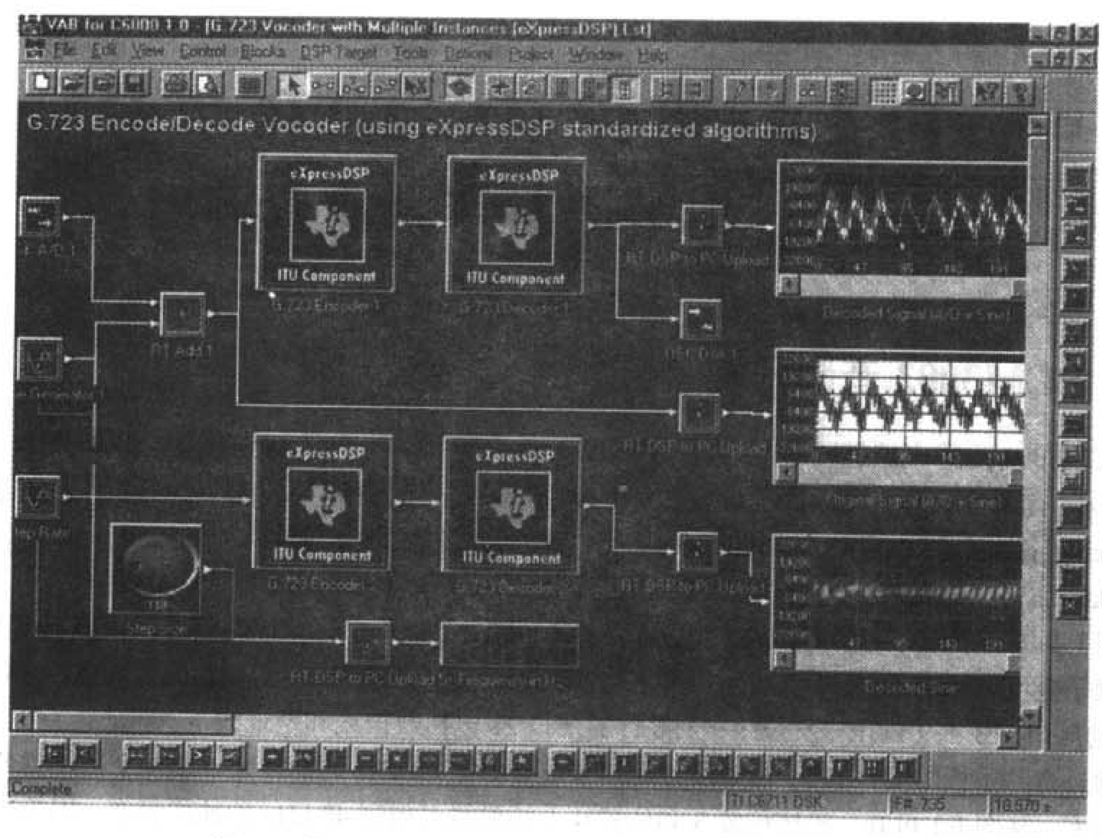


图 E.5　用 VAB 在 C6711 的 DSK 上实现 Vocoder 的框图

参考文献

1. Goldwave, at *www.goldwave.com*.
2. Cool Edit, at *www.syntrillium.com*.
3. Gram412.zip from Spectrogram, address from shareware utility with the database address *www.simtel.net*.
4. DigiFilter, from MultiDSP, at *multidsp@aol.com*.
5. R. Chassaing, *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31 DSK*, Wiley, New York, 1999.
6. R. Chassaing, *Digital Signal Processing with C and the TMS320C30*, Wiley, New York, 1992.
7. R. Chassaing and D. W. Horning, *Digital Signal Processing with the TMS320C25*, Wiley, New York, 1990.
8. Hyperception, at *info@hyperception.com*.
9. Integrated DSP, at *www.integrated-dsp.com*.
10. The MathWorks, Inc., at *www.mathworks.com*.

附录 F　用 PCM3003 立体声编解码器作为输入输出

F.1　PCM3003 音频子板

PCM3003 立体声编解码器[1,2]，是 AD535 编解码器的替代产品，它的抽样速率较高，接近 73 kHz，有两个完整的输入输出声道。和 PCM3003 一起使用的，有一个不同的通信程序 C6xdskinit_pcm.c（代替 C6xdskinit.c）以及包括原型函数的头文件 C6xdskinit_pcm.h（代替 C6xdskinit.h）。下面举几个例子来说明具有两个输入的立体声编解码器 PCM3003 的使用。

图 F.1 是一个相对比较便宜的（50 美元）PCM3003 音频子板的电路原理图，由 TI 公司提供，该音频子板可以插入 DSK 中，也可以和 TMS320C3x 接口。它和 DSK 相连时要插进 DSK 上的 80 针的接口 JP3 上（DSK 上的另一个 80 针的接口 J1 包括数据线和地址线）。

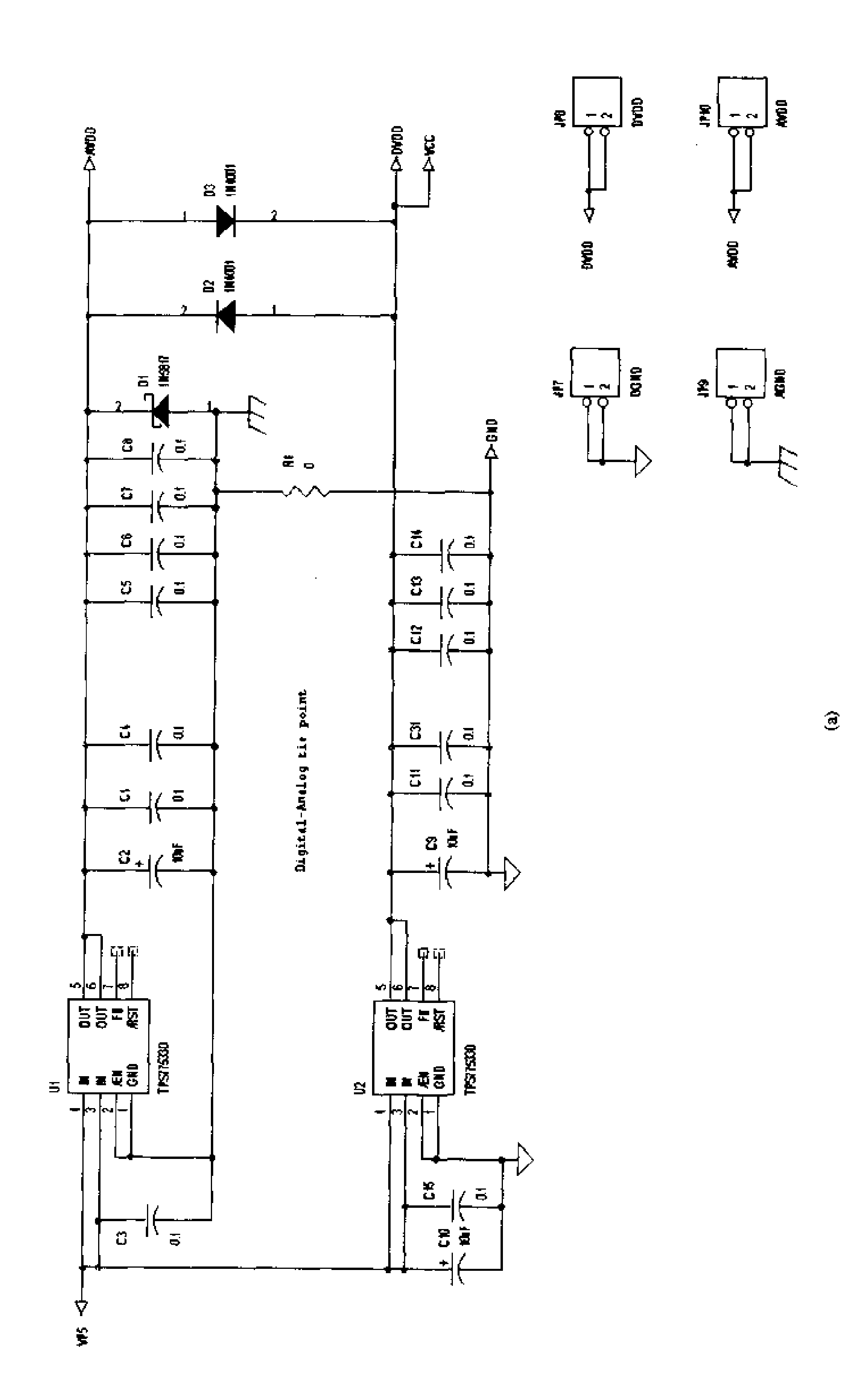
通过音频子板的接口 JP5，可以设置跳线来选择固定 48 kHz 的抽样频率或一个可编程希望的抽样速率。从图 F.1 可知，连接位置 3 和位置 4 上的跳线可得到 48 kHz 的抽样频率，因为这个跳线连接到了音频板的 12.288 MHz 的时钟上了，因此

$$F_s = 12.288\,\text{MHz}/256 = 48\,\text{kHz}$$

连接位置 1 和位置 2 上的跳线，可通过 timer 0 得到可变的抽样频率 F_s。在程序中可以设置希望的抽样频率 F_s（除非固定在 48 kHz）。F_s 是全局的，实际的抽样频率由通信支持文件 C6xdskinit_pcm.c 计算出来。表 F.1 举例说明了如何计算希望的抽样频率和相应的实际抽样频率。

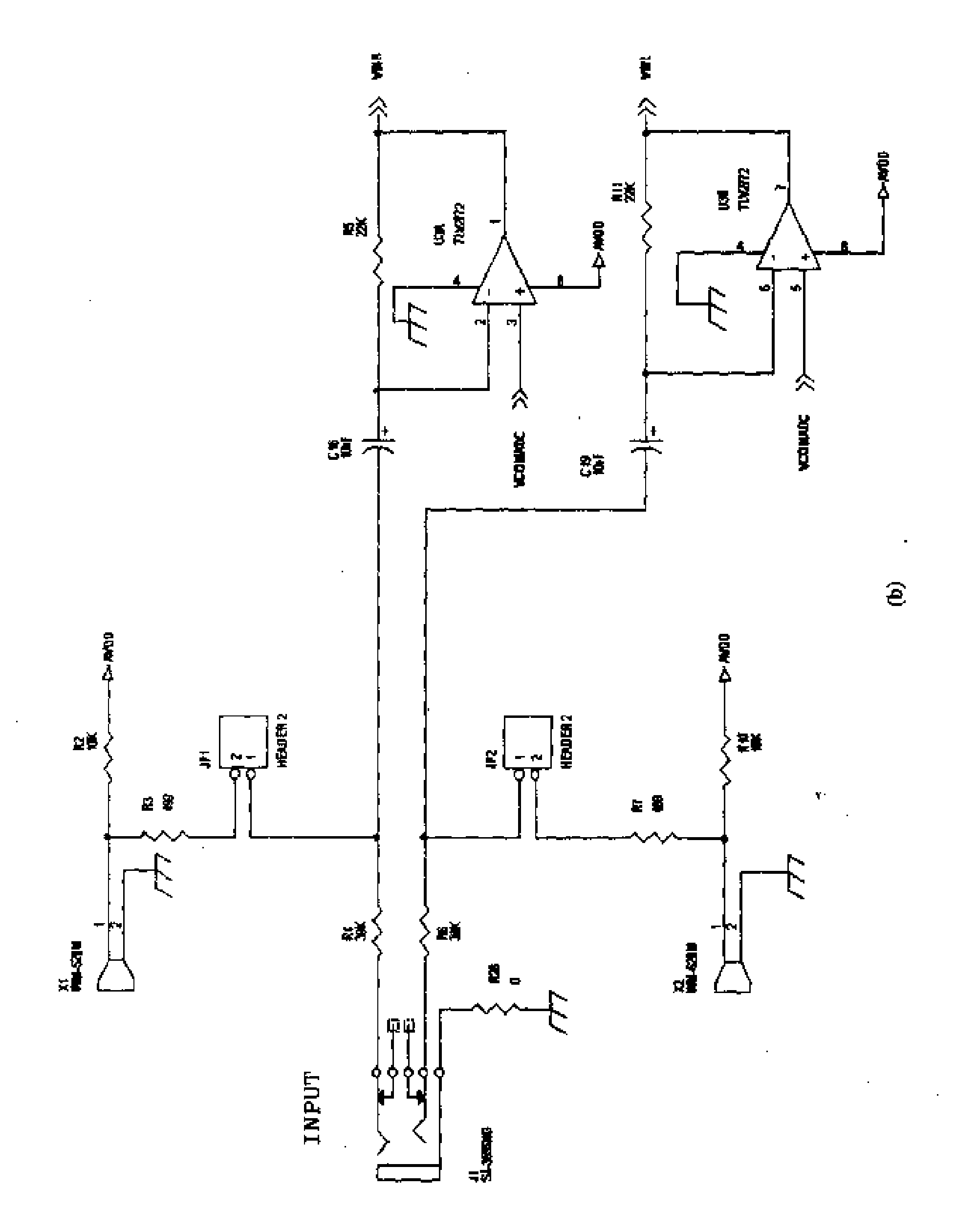
表 F.1　希望的抽样频率和实际抽样频率的对照

希望的抽样频率 F_s（Hz）	实际的抽样频率 F_s（Hz）
8000	8138.021
16000	14648.438
20000	18310.547
48000	36621.094（JP5 跳线设置在可变速率位置）
48000	48000（JP5 跳线设置在固定速率位置）
>48000	73242.187（JP5 跳线设置在可变速率位置）



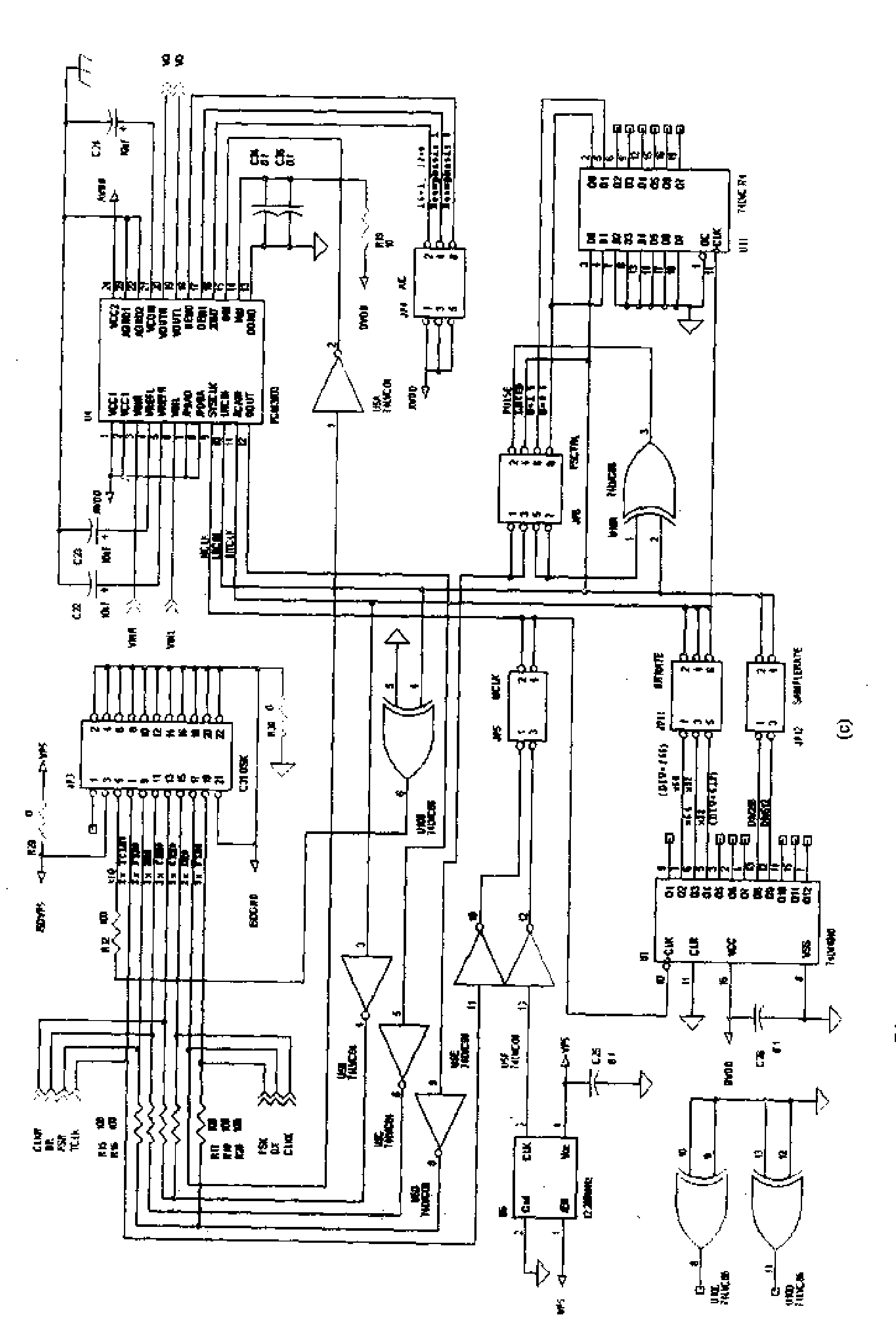
(a)

图 F.1　和 C6711 DSK 接口的 PCM3003 音频子板原理图（由 TI 公司提供）



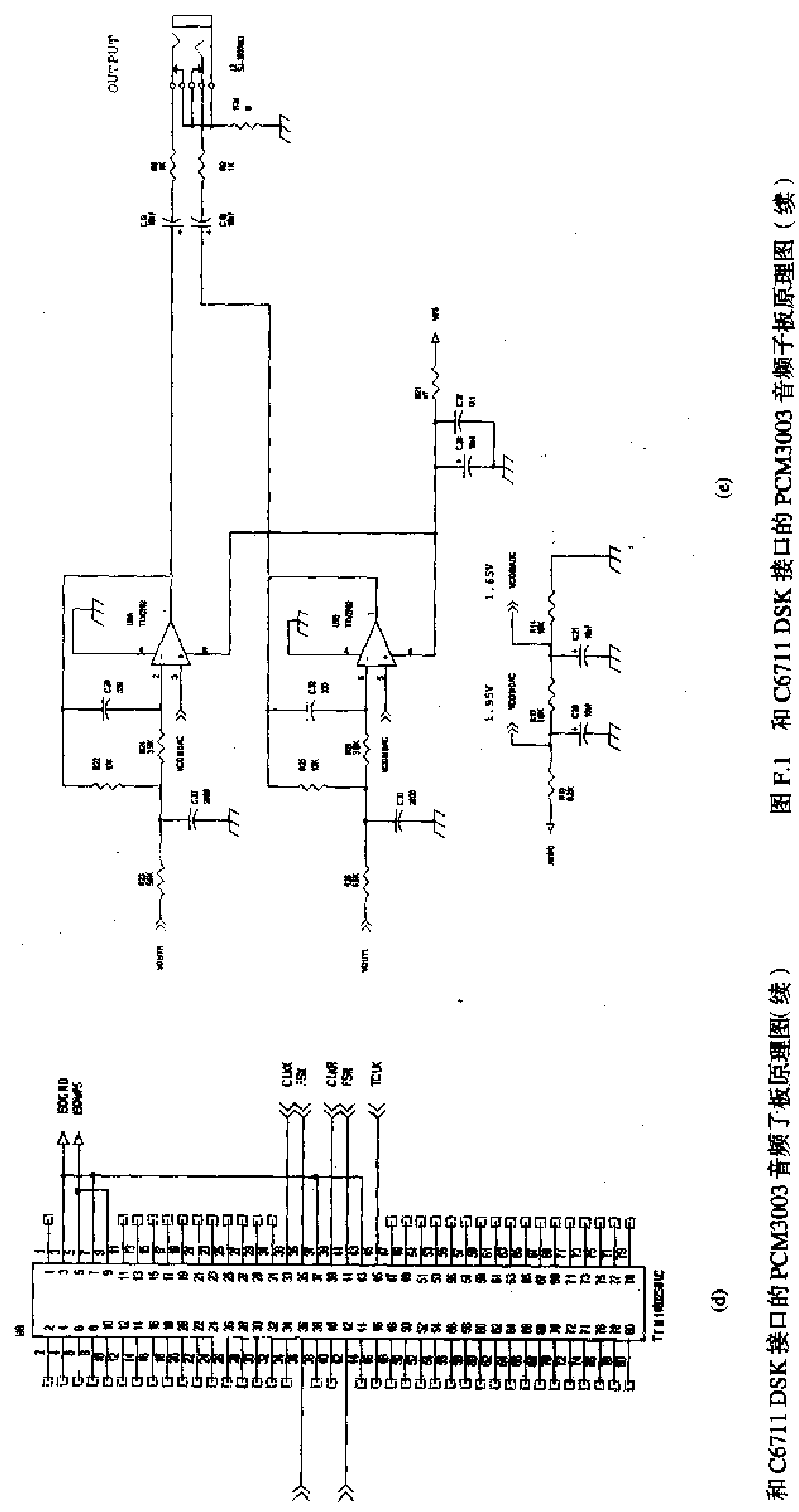
(b)

图 F.1 和 C6711 DSK 接口的 PCM3003 音频子板原理图（续）



(c)

图 F.1 和 C6711 DSK 接口的 PCM3003 音频子板原理图（续）



(d)

图 F.1 和 C6711 DSK 接口的 PCM3003 音频子板原理图（续）

(e)

图 F.1 和 C6711 DSK 接口的 PCM3003 音频子板原理图（续）

要想获得一个希望的可变抽样频率 F_s，可在程序 C6xdskinit_pcm.c 中用希望的频率（可在程序中设置）、时钟频率 150 MHz/4 和每个采样点 256 个时钟来计算得出，可得到的最大抽样频率为 73 242.18 Hz（尽管 TI 公司不推荐使用 $F_s > 48\,000$ Hz 的抽样频率）。

附录中，音频子板上的两个专用接口（立体声到单声道）用于例子里。这种接口有两个输入和一个单端输出接口，从每个输入声道中得到 16 位数据值，再从单端输出接口得到 32 位的结果数据输出（其中每个声道 16 位），这个 32 位的输出接口连接到音频板，输入到 PCM3003 编解码器上。两个输入接口中，银色的为左声道，金色的为右声道。

F.2　使用 PCM3003 编解码器的例子

例 F.1　使用 PCM3003 立体声编解码器的自环程序

图 F.2 给出了程序 loop_poll_pcm.c，该程序用 PCM3003 实现了一个环，也可参见例 2.2。本例使用板上的 AD535 编解码器实现自环。

```
//loop_poll_pcm.c Loop program with polling using PCM3003 codec

float Fs = 16000.0;                          //desired (Actual=14,648 Hz)

void main()
{
 comm_poll();                                //init DSK,codec,McBSP
 while(1)                                    //infinite loop
 output_left_sample(input_left_sample());    //IN from left,OUT from left
}
```

图 F.2　使用 PCM3003 立体声编解码器的自环程序（loop_poll_pcm.c）

可变的 F_s

在程序中，确定希望的频率 $F_s = 16\,000$ Hz，JP5 的跳线应该设置在位置 1 和位置 2 上。实际的抽样频率在 C6xdskinit_pcm.c 中计算为：

F_S =14 648.438Hz

分频器的值设置为 5（分频器的设置为整数）。

建立工程 loop_poll_pcm，工程中除了 loop_poll_pcm.c，还包含两个源文件 C6xdskinit_pcm.c 和 vectors.asm。

输入幅度大约为 1 V、频率为 1 kHz 的正弦信号，观察相应的输出是否为输入的延迟。将输入频率提高到 7 kHz 以上，验证抗混叠滤波器的带宽是否约为 6.8 kHz。

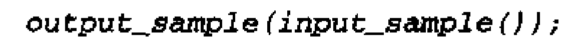
选择 View→Quick Watch 窗口来观察实际的抽样速率 F_s_*actual*，验证它是不是 14 648.438 Hz（窗口显示这个速率）。

固定的 $F_s = 48$ kHz

把 JP5 的跳线设置在位置 3 和位置 4 上，得到一个固定的抽样频率，这时在程序中设置 F_s 是不起作用。重新建立和运行程序，图 F.3 显示了用噪声作为输入时，在 HP 分析仪上显示的编解码器的输出。该图说明了抗混叠滤波器的带宽大约为 21.5 kHz。

增大输入正弦信号的幅度，当输入电压超过约 3.5 V 时，验检输出是否饱和了。

用不同声道的输入输出进行实验，如：



```
output_sample(input_sample());
```

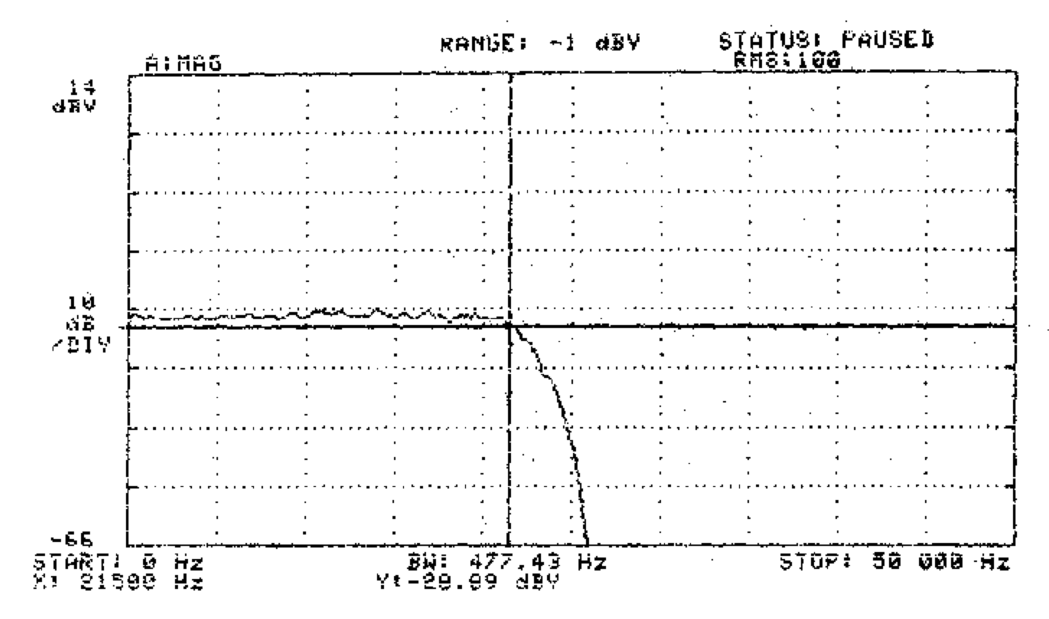


图 F.3　抽样速率为固定 $F_s = 48$ kHz，当用随机噪声作为输入时，在 HP 分析仪上显示的输出频谱（用 loop_poll_pcm）

得到一个 32 位的数据（每个声道 16 位），可用一个单声道接口输入，默认状态下用右声道（金色）作输出。左声道的接口。但是：

```
output_right_sample(input_left_sample);
```

要求立体声至单声道接口，用左声道（银色）作输入，用右声道（金色）作输出。

例 F.2　在 PCM3003 编解码器上使用中断的自环程序

该例说明了 PCM3003 编解码器采用中断驱动的自环程序。例 F.1 说明使了使用轮询的自环程序特点，也可参考例 2.1 中板上的 AD535 编解码器的使用。图 F.4 给出了实现该例的程序 loop_intr_pcm.c。

```
//loop_intr_pcm.c Loop program with interrupt using PCM3003

float Fs = 16000.0;           //irrelevant since jumper in 3-4

interrupt void c_int11()      //interrupt service routine
{
      output_left_sample(input_left_sample()); //IN/OUT from left

      return;                 //return from interrupt
}

void main()
{
      comm_intr();            //init DSK, codec, McBSP
      while(1);               //infinite loop
}
```

图 F.4　使用 PCM3003 编解码器带中断的自环程序（loop_intr_pcm.c）

建立工程 loop_intr_pcm，当 F_s 固定为 48 kHz（跳线设置在位置 3 和位置 4）时，检验是否和例 F.1 中采用轮调方式的自环程序有同样的结果。

例 F.3 给出用 PCM3003 编解码器实现 FIR 滤波器

图 F.5 显示了程序 FIR_pcm.c，该程序在 PCM3003 上实现 FIR 滤波器。例 4.4 说明了如何用板上的 AD535 编解码器实现 FIR 滤波器。滤波器的系数文件 bp41.cof 表示了一个，中心频率为 $F_s/8$（在第 4 章中使用过）。41 个系数的 FIR 带通滤波器。抽样频率设置为固定的 48 kHz。（跳线设置在位置 3 和位置 4 上）。

```
//Fir_pcm.c FIR using PCM3003 codec

#include "bp41.cof"                      //coefficient file BP @ Fs/8
int yn = 0;                              //initialize filter's output
short dly[N];                            //delay samples
float Fs = 48000.0;                      //fixed/actual Fs

interrupt void c_int11()                 //ISR
{
  short i;

  dly[0] = input_left_sample();          //newest input @ top of buffer
  yn = 0;                                //initialize filter's output
  for (i = 0; i< N; i++)
           yn += (h[i] * dly[i]);        //y(n)+=h(i)*x(n-i)
  for (i = N-1; i > 0; i--)              //starting @ bottom of buffer
           dly[i] = dly[i-1];            //update delays with data move

  output_right_sample(yn >> 15);         //output filter
  return;                                //return from ISR
}

void main()
{
  comm_intr();                           //init DSK, codec, McBSP
  while(1);                              //infinite loop
}
```

图 F.5 使用 PCM3003 编解码器的 FIR 程序（FIR_pcm.c）

建立工程 FIR_pcm，图 F.6 为在 HP 分析仪上显示的用噪声作为输入的 FIR 滤波器的频率响应，实际使用的抽样频率是固定在 48 kHz 上（设置跳线位置在位置 3 和位置 4 上得到固定的抽样频率）。从图钟可以看出，中心频率为 6 kHz，对应着 $F_s/8$。

改变跳线位置，设置成可变的抽样频率（位置 1 和位置 2），并且在程序中把 F_s 设置成 60 kHz（或者设置成任何介于 48 kHz 和 72 kHz 之间的频率）。在 C6xdskinit_pcm.c 中计算可变分频器在这个频率范围设置为 1。建立/运行该工程，并检验 FIR 是否为中心频率为 73 248/8 = 9.15 kHz 的带通滤波器。

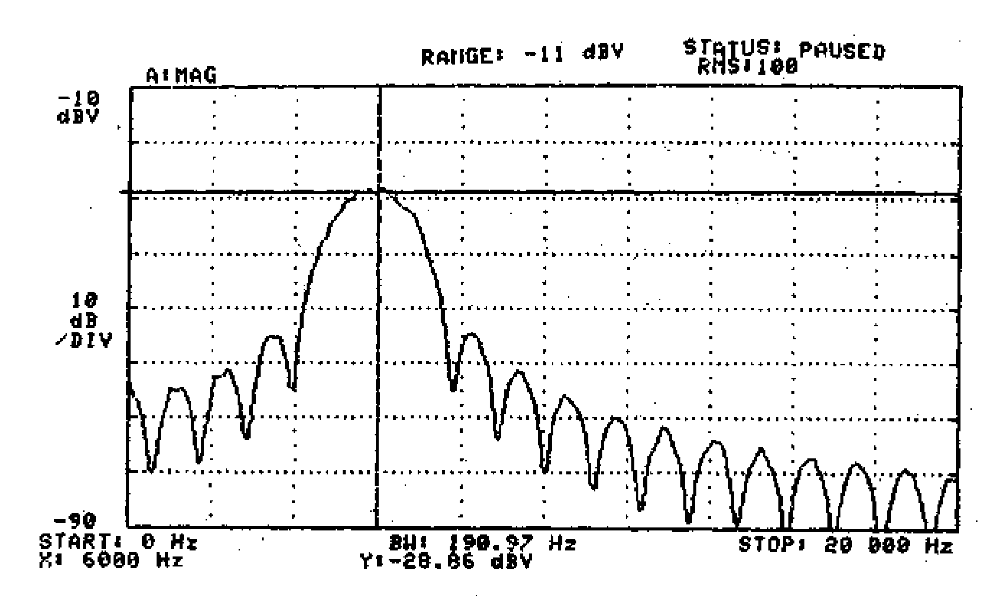


图 F.6　在 HP 分析仪上显示的中心频率在 $F_s/8$ 的 FIR 带通滤波器的输出频率响应

例 F.4　用 PCM3003 编解码器实现消除噪声的自适应 FIR 滤波器

图 F.7 给出了程序 Adaptnoise_pcm.c。该程序说明的是用 PCM3003 立体声编解码器实现消除噪声的自适应 FIR 滤波器。例 7.2 用板上的 AD535 实现了噪声消除。程序中期望的抽样频率设置为 8 kHz，但是实际的频率为 8138.021 Hz。建立工程 adaptnoise_pcm。

1. 期望的信号频率为 1.5 kHz，不期望的信号频率为 2 kHz。将期望的正弦信号（如频率为 1.5 kHz）输入到左通道，并且将不期望的 2 kHz 的正弦信号输入到右声道。运行程序，检验 2 kHz 的噪声信号是否被逐渐消除（可在程序中通过按因子 10 改变 beta 值来调整收敛速度）。打开滑动条 gel 程序 adaptnoise.gel，并且将滑动条调到 2 的位置，检验输出是否为 1.5 kHz 和 2 kHz 的两个原始的正弦信号。
2. 期望的信号为宽带随机噪声，不期望的信号频率为 2 kHz。（从 Goldwave 或噪声发生器）输入随机噪声作为期望的宽带信号输入到左声道，不期望的 2 kHz 正弦信号输入到右声道。重新启动/运行程序，将滑动条的位置调到 2 位置，图 F.8(a)是在 HP 分析仪上显示的期望的宽带信号和加性不期望的 2 kHz 正弦信号的输出频谱（滑动条的位置为 2）。图 F.8(b) 显示了不期望的 2 kHz 信号被消除后宽带信号的输出频谱（滑动条的位置为 1）。检验不期望的 2 kHz 信号是否被逐渐消除。
3. 期望的信号频率为 2 kHz，不期望的信号是宽带随机噪声。交换接口的输入，把期望的 2 kHz 信号接到左声道，不期望的随机噪声信号接到右声道，把 beta 值增加 100，重新启动/运行程序，检验随机噪声信号是否被逐渐消除（滑动条设置在 1 位置），图 F.8(c)显示了消除了不期望的宽带噪声信号后的 2 kHz 信号。

例 F.5　用 PCM3003 编解码器消除加在期望宽带信号中的窄带干扰自适应预测器

图 F.9 给出了消除宽带信号中存在的窄带干扰程序 adaptpredict_pcm，该例用 PCM3003 编解码器实现，也可参见例 7.6，它通过板上的 AD535 编解码器实现了自适应预测器。通过将 JP5 的跳线设置在固定采样速率位置，把抽样频率设置为（期望的/实际的）48 kHz。

```
//Adaptnoise_pcm.c Adaptive FIR for noise cancellation using PCM3003

#define beta 1E-10                        //rate of convergence
#define N 30                              //# of weights (coefficients)
#define LEFT 0                            //left channel
#define RIGHT 1                           //right channel
float w[N];                               //weights for adapt filter
float delay[N];                           //input buffer to adapt filter
float Fs = 8000.0;                        //sampling rate
short output;                             //overall output
short out_type = 1;                       //output type for slider
volatile union{unsigned int uint; short channel[2];}CODECData;

interrupt void c_int11()                  //ISR
 {
  short i;
  float yn=0, E=0, dplusn=0, desired=0, noise=0;

  CODECData.uint = input_sample();      //input 32-bit from both channels
  desired = (float) CODECData.channel[LEFT];  //input left channel
  noise = (float) CODECData.channel[RIGHT];   //input right channel

  dplusn = desired + noise;               //desired+noise
  delay[0] = noise;                       //noise as input to adapt FIR

  for (i = 0; i < N; i++)                 //to calculate out of adapt FIR
      yn += (w[i] * delay[i]);            //output of adaptive filter

  E = (desired + noise) - yn;             //"error" signal=(d+n)-yn

  for (i = N-1; i >= 0; i--)              //to update weights and delays
   {
      w[i] = w[i] + beta*E*delay[i];      //update weights
      delay[i] = delay[i-1];              //update delay samples
   }
  if (out_type == 1)                      //if slider in position 1
      output = ((short)E);                //error signal as overall output
  else if (out_type == 2)
      output=((short)dplusn);             //desired+noise
  output_left_sample(output);             //overall output result
  return;
}

void main()
{
  short T=0;
  for (T = 0; T < 30; T++)
   {
      w[T] = 0;                           //init buffer for weights
      delay[T] = 0;                       //init buffer for delay samples
   }
  comm_intr();                            //init DSK, codec, McBSP
  while(1);                               //infinite loop
}
```

图 F.7 在 PCM3003 编解码器上实现自适应噪声消除的程序（adptnoise_pcm.c）

建立工程 adaptpredict_pcm，（从 Goldwave 或噪声发生器）输入随机噪声作为期望的宽带随机信号，并输入 15 kHz 信号作为不期望的窄带干扰。图 F.10(a)显示了带 15 kHz 加性窄带干扰的宽带随机信号的输出频谱。图 F.10(b)显示了窄带加性干扰被消除后的输出频谱。检验 15 kHz 的干扰是否被逐渐消除了。

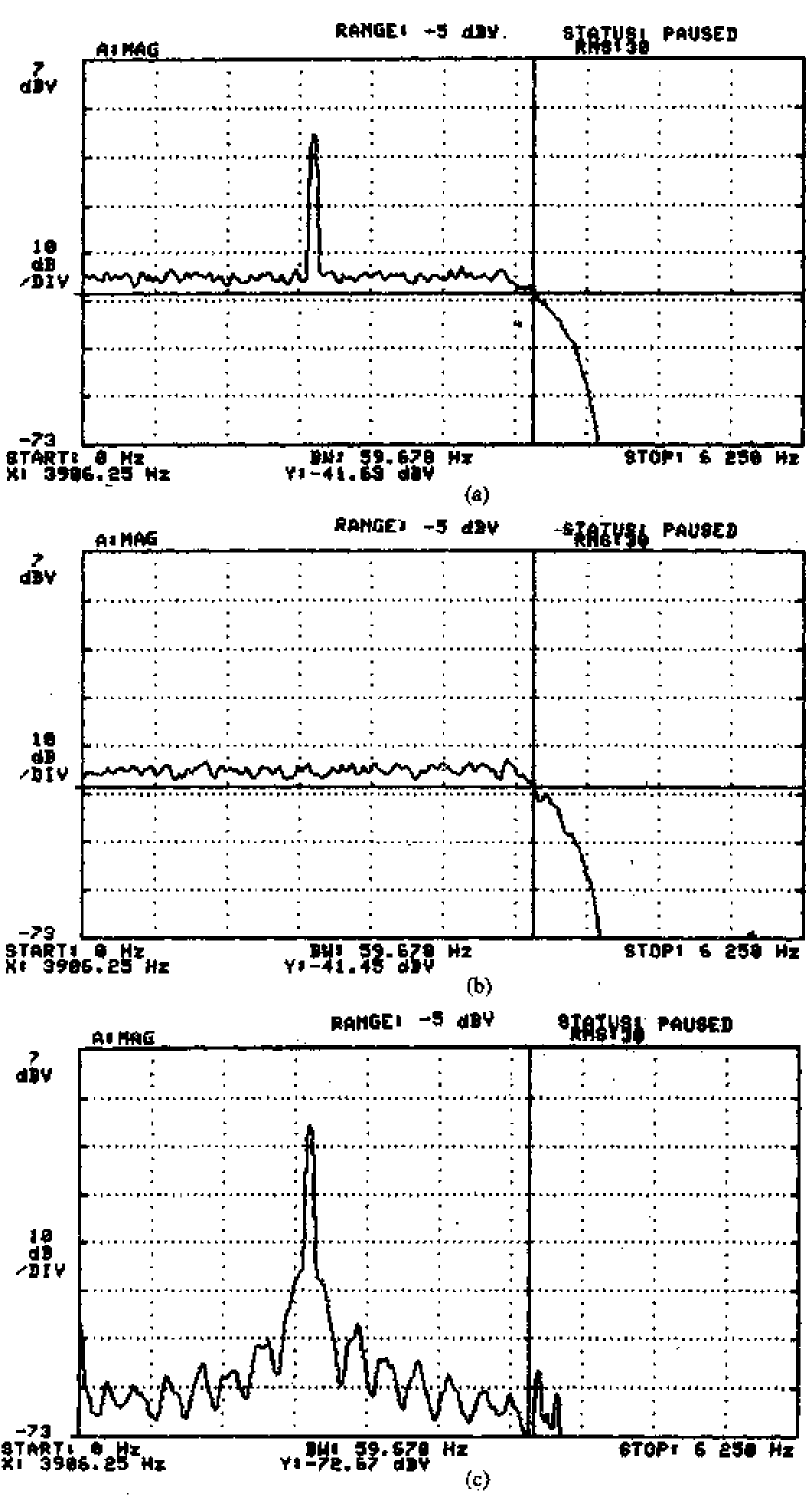


图 F.8　在 HP 分析仪上显示的输出频率响应。(a) 期望宽带随机信号，不希望 2 kHz 正弦信号；(b) 期望宽带随机信号，不希望的 2 kHz 信号被消除；(c) 期望 2 kHz 信号，宽带随机信号被消除

```
//Adaptpredict_pcm.c Adaptive predictor to cancel interference

#define beta 1E-15                    //rate of convergence
#define N 60                          //# of coefficients of adapt FIR
#define NS 256                        //size of wideband's buffer
#define LEFT 0                        //left channel
#define RIGHT 1                       //right channel
const short bufferlength = NS;        //buffer length for wideband signal
short splusn[N+1];                    //buffer wideband signal+interference
float w[N+1];                         //buffer for weights of adapt FIR
float delay[N+1];                     //buffer for input to adapt FIR
float Fs = 48000.0;                   //for fixed Fs
volatile union {unsigned int uint; short channel[2];}CODECData;

interrupt void c_int11()              //ISR
{
 static short buffercount=0;          //init buffer
 short i;
 float yn, E;                         //yn=out adapt FIR, error signal
 short wb_signal;                     //wideband desired signal
 short noise;                         //external interference

 CODECData.uint = input_sample(); //input left and right as 32-bit
 wb_signal = (float) CODECData.channel[LEFT]; //desired on left channel
 noise = (float) CODECData.channel[RIGHT];    //noise on right channel

 splusn[0] = (wb_signal + noise); //wideband signal+interference
 delay[0] = splusn[3];                //delayed input to adaptive FIR
 yn = 0;                              //init output of adaptive FIR

 for (i = 0; i < N; i++)
  yn += (w[i] * delay[i]);            //output of adaptive FIR filter

 E = splusn[0] - yn;                  //(wideband+noise)-out adapt FIR

 for (i = N-1; i >= 0; i--)
  {
   w[i] = w[i]+(beta*E*delay[i]);     //update weights of adapt FIR
   delay[i+1] = delay[i];             //update buffer delay samples
   splusn[i+1] = splusn[i];           //update buffer corrupted wideband
  }

 buffercount++;                       //incr buffer count of wideband
 if (buffercount >= bufferlength) //if buffer count=length of buffer
  buffercount = 0;                    //reinit count
 output_left_sample((short)E);        //overall output from left channel
 return;
}

void main()
{
 int T = 0;
 for (T = 0; T < N; T++)              //init variables
  {
   w[T] = 0.0;                        //init weights of adaptive FIR
   delay[T] = 0.0;                    //init buffer for delay samples
   splusn[T] = 0;                     //init wideband+interference
  }
 comm_intr();                         //init DSK, codec, McBSP
 while(1);                            //infinite loop
}
```

图 F.9 用 PCM3003 编解码器实现自适应预测的程序（adaptpredict_pcm.c）

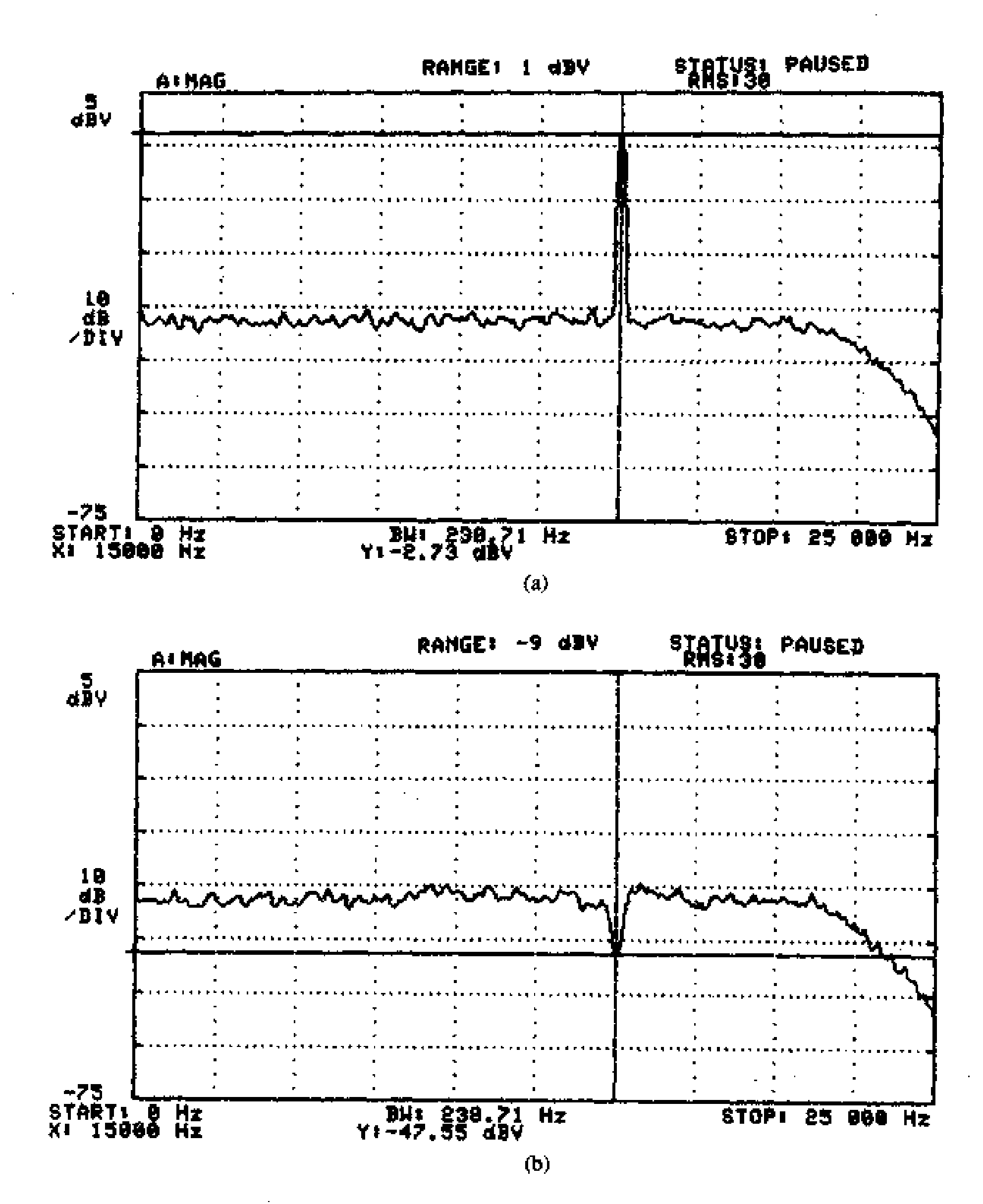


图 F.10 在 HP 分析仪上显示的自适应预测器的输出频谱。(a)期望的宽带随机信号和 15 kHz 的窄带干扰；(b)期望的宽带随机信号，15 kHz 的窄带干扰被消除

参考文献：

1. *PCM3002/PCM3003 16-/20-Bit Single-Ended Analog Input/Output Stereo Audio Codec*, SBAS079, Burr-Brown/Texas Instruments, Dallas, TX, 2000.
2. *TMS320C6000 McBSP: I²S Interface*, SPRA595, Texas Instruments, Dallas, TX, 1999.

附录 G　用于实时数据变换的 DSP/BIOS 和 RTDX

DSP/BIOS 为 CCS 提供了实时分析、列表和数据交换的功能[1-5]：当数字信号处理器（DSP）正在运行时，可以用 DSP/BIOS 来分析应用程序（目标处理器不需要停止运行）。有很多种 DSP/BIOS 应用程序接口（API）模块，用来实时分析、输入/输出等。包含在 CCS 中的 API 程序可以用来配置和控制编解码器的操作，这些 API 程序可以初始化 DSK、McBSP 和编解码器。

1. 实时分析。可以是严格的和非严格的实时分析，例如：为了使信息不致丢失，需要对输入的采样进行响应；另一方面，在两个输入样点之间的间隔时间内，把数据从数字信号处理器传输到 PC 主机中，这都是紧急的实时分析。
2. 实时列表。通过 DSP/BIOS 软件中断，可以对数据传输进行调度。可在开始就对任务/功能设置不同的优先级属性，而且可以从 CPU 执行图中得到的结果重新安排不同任务的优先级。CPU 执行图可以显示不同时刻执行的不同任务，以及判断 CPU 是否丢失实时数据。该执行图和从逻辑分析仪中得到的图相同。图 G.1 中显示了一个音频抽样（包含在 CCS 中）的执行图，该图显示了程序的执行线程，一个线程可以是 DSP 处理器执行的独立的一个指令流，可以包括 ISR 和函数调用等。不同类型的线程有不同的优先级，硬件中断（HWI）有最高的优先级，然后是软件中断（SWI），其中包括周期函数（PRD）。
3. 实时数据交换（RTDX）。当处理器运行时，通过 JTAG 接口，主机和处理器之间可以进行数据交换。RTDX 包括目标部分和主机部分，数据通过两个管道（一个用来接收，一个用来发送）传输。如果 CPU 丢失实时数据，则可以从执行图中发现。如果可能的话，可以通过重新定义优先级解决这个问题。

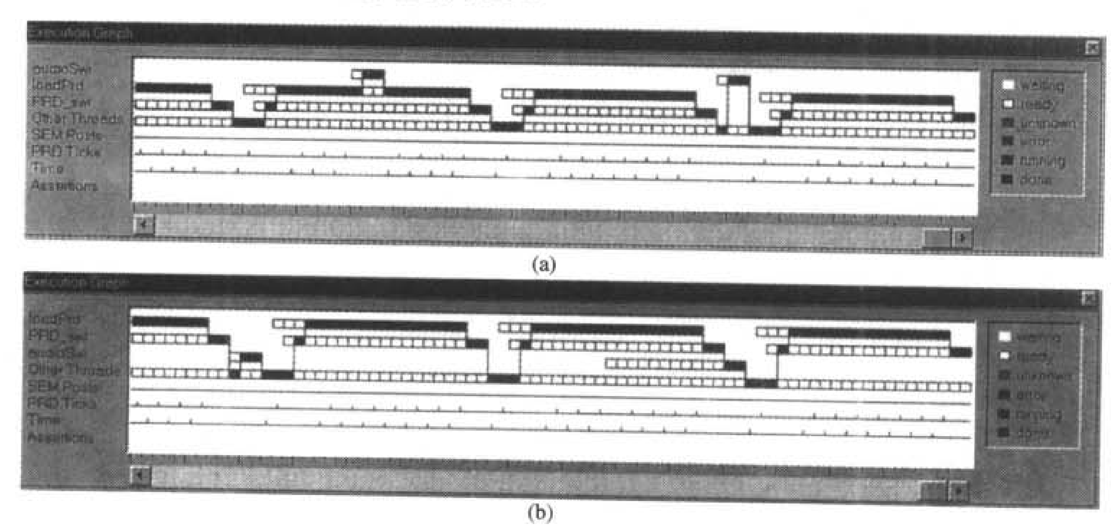


图 G.1　当 CPU 由于 NOP 过载时，CCS 画出的执行图。(a) 当设置 audioSwi 为最高优先级时，输出质量不降低；(b) 当设置 audioSwi 为最低优先级时，输出质量降低

例：DSP/BIOS的RTDX

在DSK包中有一个音频例子，它实质上是一个自环的例子，可用它来说明CPU的过载现象，过载由执行NOP指令来达到。当NOP指令增加时，就可以观测到对输出的影响。图G.1(a)显示，当任务audioSwi是最高的优先级时，它可以中断较低优先级的任务loadPrd。在图G.1(b)中，audioSwi是一个较低优先级的任务，必须等待较高优先级的任务loadPrd和Prd_swi执行完了才能执行，这种就导致数据的丢失。例如，当用音乐作为输入时，随着NOP指令的增加（上百万条），CPU开始丢失数据，此时，就可以听到输出信号的质量下降，同时执行图也显示了CPU开始丢失数据。

包含在CCS中另一个例子，使用了LOG模块的LOG_printf()函数来实时监视程序。实时库支持的C函数printf()，需要花费太多的指令周期，不能得到期望的实时监视，而LOG模块的LOG_printf()函数需要相当少的时间。尽管目标处理器和主机之间可能不是严格地实时传输数据，但LOG_prinf()函数可以严格实时地记录数据。和实时库支持处理器的printf()函数相比，DSP/BIOS支持的LOG_printf()函数的运行时间所需的指令周期只有前者的1%。

第9章中讨论了一个PLL的例子，包括与DSP/BIOS的RTDX有关的程序版本(见辅助材料)。

参考文献

1. *TMS320C6000 DSP/BIOS User's Guide*, SPRU303B, Texas Instruments, Dallas, TX, 2000.
2. *An Audio Example Using DSP/BIOS*, SPRA598, Texas Instruments, Dallas, TX, 1999.
3. *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide*, SPRU403A, Texas Instruments, Dallas, TX, 2000.
4. *Application Report, DSP/BIOS by Degrees: Using DSP/BIOS Features in an Existing Application*, SPRA591, Texas Instruments, Dallas, TX, 1999.
5. *Real-Time Data Exchange*, SPRY012, Texas Instruments, Dallas, TX, 1998.